

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

非卖品！！ 严禁（售卖和上传互联网平台）！！ 违者责任自负！！

向技术管理者

转型

软件开发人员
跨越行业、技术、
管理的
转型思维与实践

郑天民◎著



行业分析 解决方案 产品框架 业务架构 技术架构 技术创新
项目管理 交付管理 敏捷方法 过程改进 团队管理 自我管理

为从普通开发人员向技术经理、技术总监、CTO 成功转型助力、献策
一网打尽转型所需各项系统知识体系及思维方式



郑天民 网名 天涯兰

日本足利工业大学信息工程学硕士，研究方向为人工智能算法，在 SCI、EI 等国际三大索引上发表学术论文四篇，被引用达到五十余次。十年软件行业从业经验，在医疗、安防和电商行业都有所涉及，主持和参与过多个大型企业级应用和移动互联网系统的开发和管理工作，前后担任系统分析架构师、部门经理、技术总监等职务。目前就职于一家业界领先的移动医疗互联网公司，负责产品研发与技术团队管理工作。主持过十余个面向研发人员的技术和管理类培训课程，善于提炼和抽象核心内容作为教学内容，善于知识分享和技术人员培养，对架构设计和技术管理有丰富的经验和深入的理解。著有《系统架构设计：程序员向架构师转型之路》一书。

非卖品！！严禁（售卖和上传互联网平台）！！违者责任自负！！

向技术管理者

转型

软件开发人员
跨越行业、技术、管理的
转型思维与实践

郑天民◎著

图书在版编目（CIP）数据

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践 / 郑天民著. — 北京：九州出版社，2017.10

ISBN 978-7-5108-6316-5

I. ①向… II. ①郑… III. ①软件开发—工程技术人员—职业选择 IV. ①TP311.52②C913.2

中国版本图书馆CIP数据核字（2017）第262110号

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

作 者 郑天民 著

出版发行 九州出版社

地 址 北京市西城区阜外大街甲35号（100037）

发行电话 （010）68992190/3/5/6

网 址 www.jiuzhoupress.com

电子信箱 jiuzhou@jiuzhoupress.com

印 刷 北京墨阁印刷有限公司

开 本 710毫米×1000毫米 16开

印 张 19

字 数 362千字

版 次 2017年10月第1版

印 次 2017年10月第1次印刷

书 号 ISBN 978-7-5108-6316-5

定 价 59.00元

★版权所有 侵权必究★

前言

软件行业技术开发从业人员众多，但具备若干年开发经验的开发人员往往面临个人发展的瓶颈，即如何从普通开发人员转型成高层次的技术管理人员。想成为一名技术管理者，应当具备全面的知识体系，需要进行系统学习和实践，但也需要转型的思维和决心。很多开发人员有向技术管理者转型的强烈意愿，但苦于找不到好的方法和路径。本书针对“向技术管理者”这一切入点，提供技术管理所需的各方面技能和相应的学习方法，以及可能会碰到的问题和解决方法。目前市面上还没有一本完整介绍如何向技术管理者转型的书籍，本书从技术管理的定位以及如何成为一名技术管理者的角度出发，在对技术和管理工作进行深入剖析之外，还包括对行业 and 产品的思考和总结，旨在为广大开发人员提供一套系统的、全面的转型指南。

本书主要包含软件开发人员如何向技术管理者进行转型的一些思路、方法和工程实践，包括转型过程中所涉及的关于行业、技术和管理三大知识体系以及意识形态的转变和提升等内容。深入剖析成为一名合格的技术管理者所需要的各项软硬技能，重点对目前业界主流的互联网行业下所需掌握的产品开发、技术架构和技术创新领域，以及作为一名技术管理人员所需具备的组织和过程管理能力进行详细展开，并结合一些典型的场景和案例进行分析，帮忙读者了解并掌握迈向技术管理者所需的各种知识体系和实践技巧。

在结构体系上，本书共分为五大篇幅内容，包括：

1. 直面转型篇，剖析技术管理者角色，提供技术管理的维度以及开发

人员如何向技术管理者成功转型的思路。包含本书第一章内容。

2. 业务体系篇，介绍典型的互联网产品所需要的行业分析、解决方案、业务结构和产品化框架等内容。包含本书第二章和第三章内容。

3. 技术体系篇，介绍作为一个技术管理者所需要掌握的各项技术理论、架构设计方法以及如何开展技术创新活动。包含本书第四章、第五章和第六章内容。

4. 管理体系篇，从软件项目管理、研发过程体系建设、组织管理角度出发阐述技术管理者转型过程中针对团队和组织管理的各个方面。包含本书第七章、第八章和第九章内容。

5. 成功转型篇，针对转型过程介绍技术管理与意识形态的关系，以及作为一个技术管理者如何开展工作的方式和技巧。包含本书第十章内容。

本书面向立志于转型成为技术管理岗位的软件开发人员，读者不需要有很深的技术水平，也不需要具体的开发团队管理经验，但熟悉软件开发整体流程有助于更好的理解书中的内容。通过本书的系统学习，读者将在普通开发人员的基础上向前跨出一大步，在思想、方法论、实践能力和综合素质等各个方面往一名合格的技术管理者方向发展，为后续的工作和学习铺平道路。

在本书的撰写过程中，感谢我的家人特别是我的夫人章兰婷女士在我占用大量晚上和周末时间的情况下，能够给予极大的支持和理解。感谢以往以及现在公司的同事们，身处在业界领先的公司和团队中，让我得到很多学习和成长的机会，没有平时大家的帮助，不可能有这本书的诞生。

由于时间仓促，作者水平和经验有限，书中难免有欠妥和错误之处，恳请读者批评指正。

郑天民

2017 年 9 月于杭州钱江世纪城

目录

直面转型篇

1 向技术管理者转型	002
1.1 技术管理基本概念	003
1.1.1 技术管理的基本定义	003
1.1.2 技术管理演进过程	004
1.1.3 技术管理的重要性	006
1.2 剖析技术管理者角色	008
1.2.1 技术管理者角色	009
1.2.2 当技术开发碰撞技术管理	012
1.3 技术管理的维度	015
1.3.1 业务维度	016
1.3.2 技术维度	017
1.3.3 管理维度	017
1.3.4 维度关系	018
1.4 技术开发向技术管理转型	019
1.4.1 转型成功的三段式模型	019
1.4.2 转型思维导图	021
1.5 全书架构与案例	022
1.6 本章小结	023

业务体系篇

2 行业与解决方案	026
2.1 行业分析	027
2.1.1 技术管理者眼中的行业	027
2.1.2 用户研究和用户体验	031
2.1.3 商业模式分析与设计	037
2.2 解决方案	042
2.2.1 解决方案设计	042
2.2.2 解决方案示例	045
2.3 本章小结	053
3 业务结构与产品化	054
3.1 业务结构	054
3.1.1 建立业务结构	055
3.1.2 实现业务决策	058
3.2 产品化框架	061
3.2.1 技术管理者眼中的产品策略	061
3.2.2 产品化框架	063
3.2.3 产品化与项目	069
3.3 本章小结	071

技术体系篇

4 技术理论	074
4.1 软件开发理论体系	074
4.1.1 软件设计原则	074
4.1.2 技术理论的表现形式	080
4.2 架构风格	081
4.2.1 系统结构风格	081
4.2.2 数据流风格	082
4.2.3 事件处理风格	085
4.2.4 分布式风格	087
4.3 设计模式	090
4.3.1 设计模式	091
4.3.2 设计模式应用	091
4.4 架构模式	092
4.4.1 微内核模式	093
4.4.2 资源管理	095
4.4.3 服务定位	098
4.4.4 微服务架构	099
4.5 架构模型	101
4.5.1 架构视图	101
4.5.2 领域模型	105
4.6 本章小结	106
5 架构设计	107
5.1 架构设计的层次和维度	108

5.1.1	架构设计的层次	108
5.1.2	架构设计的维度	108
5.2	系统业务架构设计	110
5.2.1	系统拆分	110
5.2.2	系统集成	113
5.2.3	系统扩展	119
5.2.4	产品-项目适配型系统	123
5.3	系统技术架构设计	126
5.3.1	系统性能	126
5.3.2	系统可用	130
5.3.3	系统安全	137
5.4	本章小结	140
6	技术创新	141
6.1	技术创新概述	142
6.1.1	技术变革的基本规律	142
6.1.2	技术创新策略与模式	143
6.2	内部创新	145
6.2.1	技术内部创新的类型和要素	145
6.2.2	技术应用创新案例	146
6.2.3	技术演变创新案例	151
6.3	外部创新	155
6.3.1	技术外部创新的类型和要素	155
6.3.2	技术外部获取案例	156
6.3.2	技术跨业创新案例	158
6.4	技术知识管理	159
6.4.1	知识管理概述	159
6.4.2	技术创新与知识管理	160
6.5	本章小结	161

管理体系篇

7 软件项目管理	164
7.1 项目管理体系概述	164
7.2 需求管理	166
7.2.1 需求工程	166
7.2.2 需求建模	168
7.3 计划管理	171
7.3.1 通用计划管理活动框架	171
7.3.2 开发范围分解技术	173
7.3.3 开发工作量估算技术	175
7.4 质量管理	177
7.4.1 质量管理的维度	177
7.4.2 技术评审实施方法	179
7.5 风险管理	181
7.5.1 通用风险管理框架	181
7.5.2 软件开发与风险管理	184
7.6 交付管理	186
7.6.1 软件交付模型概述	186
7.6.2 配置管理	187
7.6.3 持续交付	194
7.7 本章小结	197
8 研发过程体系建设	198
8.1 软件过程模型概述	198
8.1.1 经典软件过程模型	199

8.1.2	管道理论	200
8.2	敏捷方法	203
8.2.1	敏捷的理念	203
8.2.2	Scrum与过程管理	205
8.2.3	精益与消除浪费	207
8.2.4	看板方法与流程管理	211
8.2.5	极限编程与工程实践	213
8.3	过程改进	215
8.3.1	CMMI中的过程改进	216
8.3.2	敏捷中的过程改进	218
8.4	建立合适的过程体系	223
8.4.1	过程裁剪	224
8.4.2	过程资产建设	225
8.4.3	轻量级过程模型	230
8.5	本章小结	237
9	组织管理	238
9.1	向下管理	239
9.1.1	理解技术人员	239
9.1.2	领导与激励	242
9.1.3	团队管理	247
9.1.4	绩效管理	253
9.2	向上管理	258
9.2.1	了解上层管理者	258
9.2.2	结果导向与目标管理	260
9.3	向外管理	261
9.3.1	政治与协商	262
9.3.2	沟通管理	263
9.4	自我管理	266

9.4.1 个人风格	266
9.4.2 处理事情	268
9.5 本章小结	269

成功转型篇

10 成为一名合格的技术管理者	272
10.1 技术管理与意识形态	272
10.1.1 思维模式	273
10.1.2 引入变化	275
10.1.3 研发文化	277
10.2 作为技术管理者开展工作	285
10.2.1 工作的层次和定位	285
10.2.2 作为推动者开展工作	287
10.3 本章小结	289
参考文献	290

直面转型篇

向技术管理者转型

软件开发人员跨越行业、技术、管理的转型思维与实践

本篇从技术管理的基本概念出发，介绍软件开发团队中技术管理者的类别和演进过程，并阐释技术管理工作的重要性。接着引出技术管理者角色，从技术管理者的活动、定位和所需技能等方面对技术管理者角色进行深度剖析，同时通过与普通程序员以及系统架构师的对比突出技术管理者角色的特定含义。

为了成为一名合格的技术管理人员，需要从技术管理的各个视角出发把握问题和安排工作，这些视角包括业务视角、技术视角和组织视角，本章同样对这些视角做了详细展开。最后，本章提出程序员向技术管理者转型所需的三段式模型，并提供了转型所需的思维导图。

本篇共有一章，作为开篇总领全书后续章节。

1 向技术管理者转型

近年来，随着软件行业，尤其是互联网行业的蓬勃发展，以电子商务、O2O、移动医疗、在线教育等为代表的互联网和互联网+化应用已经深刻影响着我们的日常生活模式。面对新的时代潮流，无论对于传统行业还是互联网行业，开发具有功能强大且用户体验好的桌面端和无线移动端应用已经成为众多软件从业人员的目标和要求。然而，现实中很多软件系统和项目研发最终都是以失败而告终。究其原因，一方面在于缺乏技术的合理应用和创新，另一方面也在于很多研发团队中技术管理存在短板和瓶颈。一个软件系统的构思、开发以及研发过程管理并不是每一个软件行业从业人员都能做的事情，需要具备对行业、技术和管理进行整合的综合能力，这种综合能力已经超出了普通技术开发人员的能力范畴。在一个典型的软件组织中，具备这种综合能力，并在行业、技术和管理领域都有专业的知识领域、丰富的实践经验以及良好的团队管理意识的人无疑是该组织的核心角色，我们把这个角色称之为技术管理者。

中国目前每年有几十万的软件开发人才缺口，同时，每年也有很多资深的开发人员面临着一种尴尬的境地。软件开发普遍被认为是一种年轻人的游戏，技术发展和演进速度过快、工作强度的日常增大、长期业务型开发工作所带来的技术瓶颈等因素都深刻影响着广大软件行业就业人员。软件行业的不稳定性、招聘信息中所规定的年龄要求、家庭和事业的平衡性迫使我们思考下一步工作的规划和个人发展方向。对于一名具备多年行业从业经验的开发人员，如果目前还处在普通的开发人员序列，还没有具备相应的意识形态和专业能力去从事技术和团队管理相关工作的话，成为一名技术管理者事实上也是自身发展所不得不面临的一个瓶颈。如何打破这个瓶颈，如何从普通的开发人员转型成为一名成功的技术管理人员，对于很多开发人员而言都可能是一个值得思考的问题。

本章围绕“向技术管理者转型”这一特定话题展开讨论。首先介绍技术管理的基本概念，然后从技术管理者这一特定角色出发，全面剖析技术管理者与普通开发人员和架构师的区别，以及对于一名技术管理者而言应该具备的核心视角。

最后，围绕“转型”问题，提出从开发人员到技术管理者成功转型所应具备的关键因素。

1.1 技术管理基本概念

现代软件系统尤其是互联网应用系统的研发具有较强的时效性，业务需求层出不穷且不断变化、技术发展和创新日新月异、团队规模从无到有快速扩张、系统的复杂性以及对行业变化的快速应变能力等成为软件开发的核心问题，如何处理这些问题成为一款产品或一条产品线成功与否的关键。解决这些问题的思路一方面来自技术，另一方面也体现在管理的必要性，而技术管理（Technical Management）的目的就是从行业和业务领域出发，通过一系列技术和管理手段在最大程度上降低软件开发的风险性，解决开发过程中存在的各种共性和特殊性问题。在深入探讨技术管理者角色之前，我们先来理解技术管理的基本含义。

1.1.1 技术管理的基本定义

在讲技术管理之前，我们先来看一下管理的概念。管理是在特定的环境下，管理者为了实现一定的目标，对其所能支配的各种资源进行有效的计划、组织、领导和控制等一系列活动的过程。所谓管理，是“管”和“理”的结合，管的对象一般是人，而理的对象是事，所以管理也可以简单抽象成人和事的综合体。

而技术管理，顾名思义，其核心概念在于两个方面，即技术与管理。技术体现在规划、设计和实现技术的能力，管理则强调组织战略和运营目标。技术与管理之间应该存在一个结合点，这个结合点就是业务体系，即技术服务于业务实现，而管理提升业务价值。技术、管理和业务构成了技术管理的三大维度，关于这三个维度之间的关系参考图 1-1。

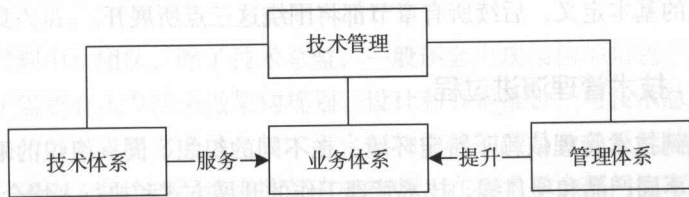


图1-1 技术管理的三个维度

更宽泛的讲，技术管理通常是指在技术行业当中所从事的管理工作，技术管理者一般同时具备较高的技术水平和管理水平，能够带领着自己所管理的团队完成某项技术任务，表现在管理者对所领导的团队在基于特定业务之下的技术规划、技术实现和技术监控。技术管理者用自己所掌握的技术知识和能力来提高整个团

队的效率，继而完成组织级别的技术任务。技术管理是技术和管理的融合，是具备较高综合层次的知识体系。

我们围绕技术管理基本概念，通过问题与解答的方式对其进一步展开：

- 技术管理的本质是什么？

技术管理是围绕技术与业务所开展的一系列活动过程，本质上体现的是一种过程体系。

- 由谁来管？

技术管理的主体是技术管理者，也就是本书所面向的主要目标对象。

- 管理什么？

管理的基本对象为人和事，但人的管理可以延伸到团队和组织的管理，而事的管理维度更为广泛，包括但不限于财、物、信息、时间等各种资源。

- 为何而管？

管理的根本目的是为了实现在组织和战略目标。

- 怎样管？

管理的职能工作是计划、组织、领导和控制。

- 在什么情况下管？

在特定环境下进行管理，即管理工作的展开需要具有一定的上下文(Context)，这种上下文可以体现为行业背景、目前的技术体系、商业目标等硬性环境，也包括如企业文化、团队组织架构等软性环境。

由于技术管理涉及面广泛，业界并没有一个关于什么是技术管理的标准说法，本书的目标也不是给出一个放之四海而皆准的定义。本书探寻的是结合目前软件行业发展的背景和方向，结合广大技术开发人员在日常工作过程中碰到的实际问题而给出的围绕“技术管理转型”这一特定主题的理解和模型，这个模型可以概括为以业务为目标、以技术为工具、以管理为手段的三段式表现形式。这是本书对技术管理的基本定义，后续所有章节都将围绕这三点所展开。

1.1.2 技术管理演进过程

上文提到技术管理依赖于特定环境，在不同的组织、同一组织的不同时期、同一时期的不同产品和项目线，技术管理工作的开展方式和执行人均会有所不同。从初创团队到一个较大规模的软件开发组织，技术管理工作以及相应的技术管理人员一般会表现为以下几种典型方式。

1. 初创团队

如果是在一个刚刚创业的团队，该团队中并没有专职产品经理和项目经理，那么开发人员可能就是团队的产品经理、项目经理或两者兼而有之。如果团队中

已经有了产品经理或项目经理，但是开发人员能力不足，那么团队就需要技术管理者来充当技术经理的角色，但本质上还是一个资深开发人员。

根据业务功能定义的范围和目标，初创团队的技术管理者实现开发计划制定、推进和管理，他可以带 1 到 2 个副手就能完成所规划功能的实现。他是主力技术开发人员，有技术难题通常也是他来亲自攻克解决。

因此，对于一个初创团队而言，技术管理者的职责很清晰，即负责核心功能的方案设计、编码实现，同时负责疑难问题分析诊断和攻关解决。

2. 小型团队

如果一个研发团队已经包含产品 / 开发 / 测试等职能小组，并且具有一条完整的主产品线，同时研发团队规模小于 20 人，那么这个团队需要的技术管理者就是一个真正意义上的技术经理。

在这样的团队中，一般已经配备 1 到 2 名资深开发人员，核心功能研发和难题攻克、进度与质量保证等工作已经可以靠他们自身能力确保得以解决。那么技术经理作为技术管理者开展的工作就应该是确保团队开发质量、生产力和专业性的提升。在对开发工作量评估、开发任务分配等团队任务管理的同时，开展代码评审、开发风险管理，进行代码模板研发与规范、最佳实践总结与推广、自动化研发生产工具研发与实施，招聘面试、新人指导、领导复盘总结改进等成为一名合格的技术管理者所应该承担的工作职责。

3. 中型团队

如果研发团队的规模超过 20 人，而且有多条稳定的产品线，团队中可能已经有多名技术经理了，那么团队就需要一名技术总监来担当技术管理者的角色。

技术总监的职责一般包括两大类，其一是组建平台级别的研发机构，搭建公共技术平台，方便各条产品线开发共用；其二通过比技术经理更高一层的职权，统一管理和协调各个产品线开发团队，确保每个产品线都应该有合格的技术经理和资深开发人员。

当发展到中型团队，除了技术总监，一般还会出现架构师角色。架构师角色的出现在于需要有人专注来做架构规划、设计和日常维护，与技术总监和技术经理相比，架构师职责更多会关注技术平台架构能够和产品业务系统的架构互相促进和支撑。通常，每个产品线可能都有架构师，在技术平台部门也有技术平台的架构师。架构师也可能带领一个团队，在有些研发团队中也会充当技术管理者。本书不对架构师角色的职责和工作开展方式做过多讨论，读者可参考相关资料^[1]对系统架构设计和架构师角色进行深入理解。

4. 大型团队

当团队中出现多个技术总监，团队也不断成长到 100 人或更大规模时，需

要的就是 CTO。关于 CTO 的职责和管理工作边界也存在一定的异议，有些公司的 CTO 管理的仅仅是技术研发团队，而有些公司则统领技术、产品和运营等团队。有些大型公司有分管产品的产品副总裁，也有分管技术的技术副总裁，而且把技术副总裁叫 CTO，产品副总裁叫产品 VP，这实际上并不一定合适。根据我们在上一节中讨论的结果，技术管理是业务、技术和管理三者的结合体。真正的 CTO，对于软件产品和技术需要做到全面把控，以实现商业、产品、技术、管理、团队相平衡的综合目标。

CTO 的职责一般表现为四个主要方面。首先是目标业绩导向，借助于洞察客户需求、捕捉商业机会、规划技术产品等手段，通过技术产品领导业务增长，有清晰的战略规划、主攻方向，带领团队实现组织目标。其次，关注前沿平台，在大型研发团队中，一般都会有专门的团队做技术应用创新探索和前沿技术预研。而且要和技术平台团队、应用研发团队形成很好的联动作用，让创新原型能够很平滑地融入商业平台并在应用研发线得到规模化应用。再次，梳理研发过程，CTO 需要站在全局立场来端到端改进业务流程，为业务增长提供过程支持。最后，重视组织与人才建设，公司文化和价值观的传承、研发技术和管理族团队梯队建设、创建技术创新机制、激励研发人不断向前发展、提高团队整体战斗力是这方面的具体表现形式。

以上关于技术管理的讨论通过表 1-1 可以清晰展示其演进过程。显然，不同的技术管理表现形式对于技术管理者的要求也是不尽相同，即技术管理者也具有层次性。本书讨论的技术管理以及相应的技术管理者角色更多是站在大、中型团队的角度看问题，即以技术总监和 CTO 为目标群体，提供一整套完整的方法论和工程实践以满足向目标技术管理者的转型需求。

表 1-1 技术管理演进过程

团队阶段	团队规模	团队特点	技术管理者角色描述
初创团队	10人以下	一人身兼数值，有一个产品	核心开发人员和问题解决者
小型团队	10~20人	具有基本分工，产品线成型	开展过程建设
中型团队	20~100人	具备产品线和产品平台	研发平台建设、统一资源调配
大型团队	100人以上	业务和组织结构非常复杂	整合业务、技术和管理，对组织目标负责

1.1.3 技术管理的重要性

技术管理的本质目的是为了**实现组织和战略目标**，其作用恰恰也体现与此。相较以前的传统开发模式，以互联网应用为代表的软件产品开发模式已经发生了

巨大变化。伴随着这些变化，更为高效的产品开发过程将具备更大的竞争优势，这就迫使我们不得不提升组织的技术管理水平，因为技术管理能在以下四个方面促进组织和战略目标的达成。

1. 正确的产品

成功的产品自然需要整个组织级别的规划和实施，这其中战略、产品、运营和技术等各个维度缺一不可，但首当其冲的是对行业和业务的理解，确保产品方向的正确性。作为一名技术管理者，同样需要参与产品的战略规划和实施过程，从技术角度给出解决方案。技术管理将在业务结构的确定、产品化策略和实现平台的建立工作中发挥作用。关于业务和产品化的研发方法与工程实践我们将重点在第二篇中具体展开讨论。

2. 创新的产品

对于技术管理而言，通过技术创新开发出具有市场领先水平的产品有助于最终战略目标的达成。毫无疑问，第一个进入市场的产品将会有助于增强该产品的市场占有率。这里的“第一个”可以表现为第一个对市场机遇做出反应、第一个采用某种新技术，或者说在同等技术能力条件下，比别人做出更快的市场反应，也即意味着创新。创新可以是业务的创新，也可以是技术的创新。

在互联网行业中，时机可能比任何其他因素更为重要。在一些行业中，市场窗口只会开放很短的一段时间。在这种背景下，产品能不能成功很大程度上取决于产品投放市场的时间。如果在同等产品规划和运营策略下，也即在相同的业务创新条件下，技术创新就会成为影响产品成功的决定性因素。技术创新能够在改善产品用户体验和缩短产品研发生命周期上提升产品成功的概率，如图 1-2 所示，通过缩短开发时间从而快速推出新产品能带来产品收益上的增长。而对于互联网产品而言，很多时候错过产品发布时机就意味着再也没有机会。关于各种主流技术的应用以及技术创新的类型和实施方法我们将重点在第三篇中具体展开讨论。

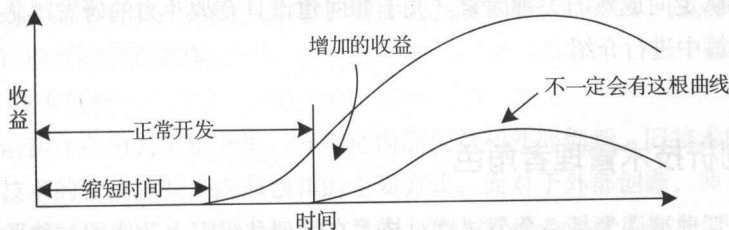


图 1-2 通过技术创新缩短同行业产品开发周期

3. 高效的研发过程

软件行业大多数产品开发由时间和成本决定其投入，即一定数量的开发人员通过一定时间的开发工作完成某个具体产品。显然，开发周期的缩短同样意味着

开发成本的降低，因此开发成本与开发周期密切相关，产品开发周期时间与成本之间并不是一种简单的线性关系，随着开发时间的增长，开发成本增长的趋势越来越明显。出现这种现象是因为软件开发过程中对范围变更的控制、计划的监控、资源的合理安排都存在风险，且风险随时间演变其发生的概率和造成的影响就越大。

通过建立高效的研发过程体系可以提高开发效率，通过提高开发效率而节省下来的资源可以再投入到新产品的开发中去，从而使更多的新产品投向市场，或者减少新产品开发的总体成本。对于软件开发这个特定行业而言，减少浪费是提高开发效率的首要切入点。浪费分成纯粹的浪费和必要的浪费，其中纯粹的浪费需要消除，而必要的浪费可以进行压缩。需要结合日常研发过程，对如何识别这些浪费、如何消除纯粹的浪费以及如何压缩必要的浪费需要进行剖析。技术管理的作用通过项目管理的实施方法以及研发过程体系建设等维度发挥其作用，这部分内容我们将重点在第四篇中进行介绍。

4. 成熟的研发团队

研发效率的提升需要过程，但也需要充分考虑人的因素，因为最具有创造性的技术人员几乎总是最有效率。这一点在以创新能力和知识型技能为前提的软件产品开发中尤为突出。当前的互联网产业在高速发展的同时也伴随着浮躁的行业环境，产品开发人员往往表现出较大的流动性，通过技术管理手段打造一个稳定、健康的研发环境有助于吸引优秀的研发人才。反之，如果长时间处于一个疲于奔命的环境中，研发人员会感到失望和痛苦，从而不可避免去寻找新的工作环境。

一旦组织已经存在一个高效的产品决策、设计和开发过程体系，剩下的就是更好的管理和留住关键的开发人员，这就涉及组织管理范畴的内容。除了向下管理之外，组织管理还需要做到合理的向上管理和向外管理。当然，技术管理者作为一个团队的核心人物，在个人风格和处事能力上的不断自我管理和提升也是促进研发团队走向成熟的关键因素。关于如何建设具有战斗力的研发团队我们将重点在第五篇中进行介绍。

1.2 剖析技术管理者角色

技术管理被认为是一个公司针对技术在管理和组织上达成的一种平衡机制。正如 1.1.1 节中所述，管理是指导和控制组织进行相互协调的活动，而这里的组织泛指职责、权限和相互关系得到安排的一组人员及设施，所以技术管理就成为指导和控制组织中关于技术的相互协调的一系列活动。技术管理活动与公司的战略管理活动以及针对特定团队的组织管理活动都有直接交集，连接着一个公司中

最为重要的两个核心领域。

以技术总监、CTO 等角色为代表的技术管理者是技术管理活动的执行者，技术管理的平衡机制要求给技术管理者带来了挑战，意味着技术管理者需要同时具备处理两个核心领域的能力，即技术管理者需要能够从公司的战略目标出发构建满足业务发展中的技术体系，同时又能够站在组织角度设计满足产品研发过程的研发体系，最终实现技术在管理和组织上的统一。

1.2.1 技术管理者角色

角色是规定一个人活动的特定范围以及与人的地位相适应的权利义务和行为规范，是组织对一个处于特定地位的人的行为期待。角色定位需要认清自己以及竞争者的位置和职责，正视自我，同时以强烈的职业意识给自身的事业和未来发展确定一个方向。我们从所从事的活动、定位和所需技能对技术管理者这一角色进行全面剖析。

1. 技术管理者的活动

技术管理者是负责规划、设计和实现技术能力，从而完成组织战略和运营目标的人。通常，这个角色需要完成以下几项活动：

（1）技术预测

技术预测的需求来自于行业和业务的快速发展。根据业界成熟的技术实现体系，能够在一定程度上预判未来几年的技术发展趋势，是技术推动和反哺产品演进的一个重要方面。当然，技术预测的前提是对当前所从事行业的深入理解，能够设计和实现业务的技术解决方案。

（2）产品研发

产品研发同样关注于业务体系，通过对业务体系的了解建立业务架构，并发挥技术在实现业务决策上的能动性。另一方面，明确产品发展策略，通过建立技术平台推动产品平台和产品线，并基于项目化手段实现产品研发，这点可以说是作为技术管理者的主要工作。

（3）技术创新

技术创新存在两大主要分类，分别是内部创新和外部创新。旧技术的新应用以及现有技术的自我演变是内部创新的主要方式，而对于外部创新，通过技术合作和跨业创新同样可以达到技术创新的目的。技术管理者无疑是技术创新的主要推手。

（4）技术标准

技术标准的范畴可以很广，在软件行业，现有技术的应用方式、软件的交付质量、版本的发布模式等都可以算是组织级别技术标准的一部分。技术标准的制

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

定需要同时考虑技术、管理和组织等几方面因素，技术管理者需要根据自身团队的特点追求适合当前环境的标准化体系。

(5) 过程建设

软件开发是一个系统工程，涉及软件工程、项目管理、系统交付等多个维度，也需要对软件过程模型进行裁剪，并应用过程改进思想和工程实践不断推动过程体系的合理化发展。在现代软件开发过程中，技术管理者需要建立合适的、轻量级的过程体系并实现过程资产建设。

(6) 组织交流

在一个组织，尤其是大型组织中，技术团队与市场、销售、产品、运营一样同属于组织的一个组成部门，很难说技术可以独立于其他团队进行自我发展，因为技术服务于公司的战略目标、服务于业务和产品发展需求。从这个角度讲，技术在组织中的地位通常不会太高。通过各个团队之间的有效沟通和交流，可以让其他团队更好的认识以及理解技术团队的工作方式和能力范畴，避免交流不畅所形成的团队壁垒。另一方面，技术团队内部同样需要交流，技术分享和培训、绩效沟通等也属于这一范畴。

(7) 日常管理

除去以上六点主要工作内容，技术管理者的活动还包含各种日常事务的处理。日常管理的接口可以是对外、对内，也可以是对上、对下，同时针对技术管理者自身的自我管理也是属于日常管理的一部分。

2. 技术管理者的分类

(1) 管理者角色

管理学大师亨利·明茨伯格(Henry Mintzberg)研究发现管理者扮演着十种角色，这十种角色可被归入三大类：人际角色、信息角色和决策角色^[2]。人际角色归因于管理者的正式权利，管理者所扮演的三种人际角色是代表人角色、领导者角色和联络者角色。在信息角色中，管理者负责确保和其一一起工作的人能够得到足够的信息。而在决策角色中，管理者处理信息并得出结论，管理者负责做出决策，并分配资源以保证决策方案的实施。管理者十大角色分类详细描述参考表 1-2。

表1-2 亨利·明茨伯格管理者角色

类型	角色	角色描述
人际角色	代表人角色	作为象征性的组织头脑必须行使一些具有礼仪性质的角色
	领导者角色	和员工一起工作并通过员工的努力来确保组织目标的实现
	联络者角色	与组织内个人、小组一起工作、与外部干系人建立良好的关系所扮演的角色

续表

类型	角色	角色描述
信息角色	监督者角色	持续关注内外环境的变化以获取对组织有用的信息，接触下属或从个人关系网获取信息，依据信息识别工作小组和组织潜在的机会和威胁
	传播者角色	分配作为监督者获取的信息，保证员工获取必要信息以便切实有效完成工作
	发言人角色	把信息传递给单位或组织以外的个人，让干系人得到了解
决策角色	企业家角色	作为监督者发现的机会进行投资以利用这种机会
	资源分配者角色	决定组织资源（财力、设备、时间等）用于哪些项目
	冲突管理者角色	处理组织运行过程中遇到的冲突或问题
	谈判者角色	进行必要的谈判，对象包括员工、供应商、客户和其他工作小组，以确保小组朝着组织目标迈进

（2）技术管理者的角色分类

技术管理者显然符合上文中的通用管理者的角色和定位，但结合软件行业的特征，我们发现还是可以把技术管理者所扮演的角色进行分类，这些分类实际上也是对技术管理者角色的一些认知，包括但不限于：

· 规划型

明确公司整体战略，能根据整体战略目标分解到技术团队的职责和目标，并根据目标制定具体的执行计划和实施方法。

· 执行型

对于已经确定的目标，明确团队及各岗位成员的职责，并传递给每一位执行者，研发执行技术标准，并能履行岗位职责。

· 问题解决型

针对执行层上报或通过自身观察所得到的问题，分析、查找问题根源，设计方案并申请所需要的资源。

· 监督型

不断建立和完善技术研发体系，严格执行研发过程。对过程和结果提出自身的理解和判断，从事情出发追求结果。

· 领导型

具备良好的领导风格和个人魅力，通过自身思维影响并决定团队的思维，通过自身的风格影响并决定团队的风格。

· 教练型

促进团队成员知识和工作能力的不断提升，善于针对每个人以及整个团队找

到瓶颈所在，并培养团队成员发现问题和解决问题的能力。

通常，一个技术管理者会同时属于上述几种分类中的多种或全部。本书所阐述的技术管理者角色也会同时包含所有这些分类所具备的基本属性。

3. 技术管理者的技能

作为一名合格的技术管理者，完备的技术领域知识是必备的技能，同时也应该包括在 1.1.2 节技术管理演进理论中提到过的各个不同阶段所应具备的各项技能。我们对这些技能进行再次分类和梳理，可以抽象成以下三个方面：

（1）业务和行业理解能力

在软件应用系统开发过程中，业务架构驱动技术架构现象非常普遍。提升业务领域知识和提升技术领域知识一样，都对产品研发有直接的影响。从这个角度讲，技术管理者同样需要深入特定行业和业务体系，具备从业务到技术的正向跨领域的技能。

（2）技术实现和创新能力

技术管理者管理的是技术，技术可以分成技术应用和技术创新两种主要方式。技术管理者在应用主流的技术体系之外，还需要考虑结合业务和产品的技术创新。从这个角度讲，技术管理者需要从技术反哺业务，具备从技术到业务的反向跨领域的技能。

（3）组织管理能力

无论是传统型软件还是互联网应用，当前的开发模式已不再崇尚靠能力出众的个人来决定系统的产出，而是要靠团队。技术管理面临着项目计划同步、第三方服务集成、外部团队协作等团队性活动需求，很多场景下管理者需要与内部团队、外部团队统一协作才能设计出适合业务发展方向的产品。从这个角度讲，技术管理者应该具备跨越多个团队的能力。

本书后续章节也会从不同的维度对这些技术管理者所需的各项技能进行展开讨论，确保从开发人员到技术管理者的成功转型。在此之前，我们再来看一下技术开发和技术管理工作之间的区别和联系。

1.2.2 当技术开发碰撞技术管理

技术管理者的主要工作围绕技术和管理这两个维度展开，通过对技术和管理两个维度的排列组合，我们可以得到技术与管理的四种组合结果。本书介绍是如何成为技术能力和管理能力都比较出众的技术管理型人才的方法，而主要面向的对象是目前技术能力出众但管理能力有待进一步提高的技术型人员。技术型人员普遍存在两种角色，一种是普通的程序员，一种是代表技术专业度较高的架构师。接下去我们针对这两种技术角色与本书讨论的技术管理角色做一下比较。

1. 对比程序员和技术管理者

当程序员遇到技术管理者能否碰撞出火花？我们从程序员的特点和技术管理者的要求两方面出发做一下对比，对比项包含以下 12 项。

（1）个体与组织

程序员更多的是关心个体的事情，关注自身工作是否顺利完成；而技术管理者更关注是否充分挖掘了每个员工的最大价值，团队成员是否已经尽了自己最大的努力，在管事的同时更要管人。

（2）完美与成功

程序员在工作上常常习惯从深处、细微处着手，追求完美；管理者则并不苛求每一个细节都完美，主要是宏观把握，因而很多事情处理起来显得不够精细和具体。

（3）是与非

程序员坚持非黑即白的是非观，由于长期从事技术活动，更相信经过反复验证的事情，是非权责分明；管理者则认为在管理中没有绝对的正确或错误，只要不违背一些特定的原则即可，所以他们在观念上也就没有非黑即白，而是因人而异，具体问题具体分析。

（4）人与事

程序员做事只对事不对人，无论对客户还是上司，就事论事，不会变通，更不会因人而异；管理者则坚持对事也对人，常常根据具体情况做出对应的反应。

（5）过程与结果

程序员更多的是享受过程的乐趣，对于结果并不过分重视，只管耕耘；管理者则更强调工作的结果和价值所在。

（6）加法与乘法

程序员在工作方式上是“算加法”，通常是完成一件事再去做另外一件；管理者则是“算乘法”，所有关键要素齐头并进，其中任何一个要素都要求配合完成。

（7）收敛与发散

程序员倾向于收敛思维，思考问题时思维模式单一化、机械化；管理者则是发散思维，强调融会贯通。

（8）指标与理念

程序员做每件事都需要量化的指标，坚持科学的观念；管理者更多依靠的是与“科学”相对应的概念性的理念。

（9）技术与业务

程序员在完成开发工作的同时，很大程度上追求的是技术的突破，不善于从干系人角度出发理解业务并进行抽象；而管理者更多关注业务体系的建立、产品

平台的构建以及组织战略的最终达成。理解干系人的业务痛点，并基于业务需求进行技术规划和实现是其基本工作内容之一。

（10）推动与服从

程序员独立思考，有好的想法但并不喜欢分享，更加不会作为推动者去主动落实这些想法；而技术管理者应具备领导力与推动力，除了技术演进，在团队价值取向、组织趋势把握、组织运营和人才发展等各个方面都可能需要发挥其主导性。

（11）散乱和集成

程序员拼命工作而不是聪明的工作，因为缺少系统化的工作规程；而技术管理者需要从系统工程角度出发，对软件开发、项目管理和过程改进等系统过程进行合理规划，形成统一的工作模式，确保团队成员都能在同一节奏上开展开发工作。

（12）主观与客观

程序员，尤其是年轻的程序员的一大特征就是情绪化思维，对碰到的问题倾向于使用主观意识去寻找方法，同时又有一些顽固，钻牛角尖的场景并不少见；而技术管理者通常具备全面的思考和分析模式，倾向于使用换位思考从问题的内因、外因出发，找到团队内部和外部能够解决问题的资源，确保问题得以高效解决。

2. 对比架构师与技术管理者

系统架构师的职责和主要工作在于识别干系人并让他们参与进来；理解和记录系统功能和非功能相关的关注点；通过需求分析，架构师梳理并抽象系统的各项功能性和非功能性需求，并对这些需求进行系统建模；创建并拥有应对这些关注点的架构定义，对功能性和非功能性需求，从扩展性、性能、可用性、安全性、伸缩性等架构设计的基本要素出发定义架构；同时在把架构实现为具体系统的过程中起主要作用。

就一个完整的系统开发生命周期而言，架构设计活动有其时效性。图 1-3 体现了传统瀑布模型下的系统开发生命周期与架构师参与情况，从图中可以看出在由需求分析和系统建模所构成的系统初始阶段以及由服务集成和产品接受所构成的最后交付阶段，架构师会较多的参与到系统建设过程中去，具体参与程度取决于系统本身的特征以及生命周期模型。

而就一个完整的产品开发生命周期而言，技术管理活动也具有其时效性，这种时效性相较于系统架构设计和实现等技术专业类活动而言还具有较大的灵活性。我们可以理解为系统开发生命周期是整个完整软件产品生命周期的一部分，如图 1-4 所示。在系统开发工作开展之前，技术管理者需要进行行业分析、技术解决方案的设计以及产品开发策略的规划，同时针对行业特点也可能会从事部分

的技术预测工作。而在系统开发工作结束之后，随着产品和运营工作的开展，技术管理者也要深入其中从组织战略的角度出发对技术提出进一步的规划方案和创新措施。

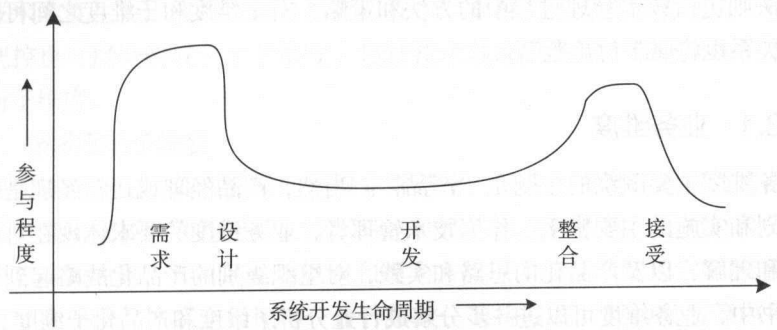


图1-3 系统开发生命周期与架构师参与情况

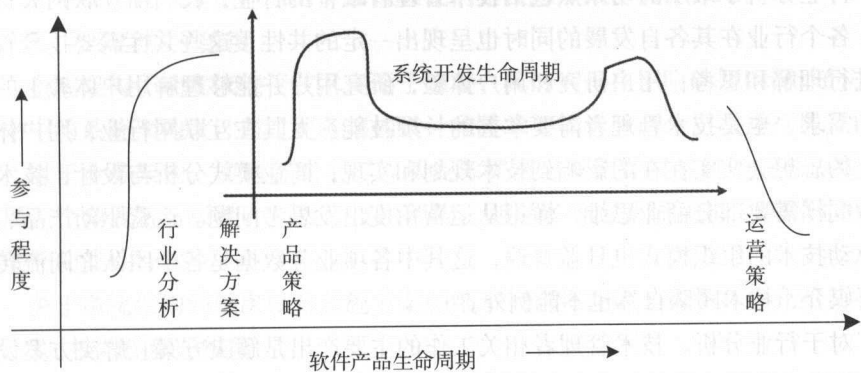


图1-4 软件产品生命周期与技术管理者参与情况

1.3 技术管理的维度

技术管理无疑是一项综合性工作，包含业务、技术和管理这三大维度。如何把这三大维度通过一定的方式展示给开发人员，从而让开发人员能够快速和准确的理解进而完成转型是我们首先需要明确的一个核心问题。针对该问题，很难通过一个面面俱到的模型或视图做出完整的回答。我们在思路也是使用分解的策略，把技术管理者所应该具备的转型方法转换为不同的子维度。子维度的提出也是为本书如何进行内容组织提供了具体的表现形式。

要想清晰、明确的展示技术管理维度，首先要对维度进行建模，即通过统一的表述方式和模型来展示不同的维度。在本书中，每一个维度将从切入点和实施

方法两个角度出发进行展开讨论。其中切入点描述某一个维度特定的关注点，并提供切入该维度的方向，任何一个维度都应该具备若干与其他维度不一样的切入点，以便在技术管理过程中能够找到问题所对应的切入点并进行优化和管控。而实施方法则包括技术管理过程中的方法和策略，对于维度和子维度之间可能存在的相互关系也将属于讨论范畴。

1.3.1 业务维度

业务维度主要围绕的主题是“产品”。当然，产品经理或运营经理是对产品进行规划和实施的主要推手。作为技术管理者，业务维度的要求体现在对于行业的分析和理解，以及产品化的思路和实践，对组织级别的产品化战略起到促进作用。本书中，业务维度可以进一步分解成行业分析子维度和产品化子维度。

1. 行业分析子维度

行业分析子维度的切入点包括技术管理者眼中的行业，在当前互联网大背景下，各个行业在其各自发展的同时也呈现出一定的共性，这些共性需要技术管理者进行理解和思考；用户研究和用户体验，研究用户并能够理解用户体验上的痛点和需求，也是技术管理者需要掌握的一项技能。尤其在互联网行业，用户体验至上的思想会实实在在的影响到技术规划和实现；商业模式分析与设计，技术管理者同样需要部分商业思维，懂得从运营角度出发思考问题。运营驱动产品、产品驱动技术的组织模式也日益普遍，这其中各项业务数据是各个团队之间流转的主要媒介，技术团队自然也不能例外。

对于行业分析，技术管理者相关工作的主要产出是解决方案。解决方案设计是一项综合性的工作，具备基本设计思路的同时还需要考虑某个行业的特定需求。同时，技术也是解决方案的一个重要组成部分，技术可能是各行业通用的表现形式，也可能因行业而异。

关于行业和解决方案的详细内容我们将在第2章中具体介绍。

2. 产品化子维度

产品化子维度的切入点主要包括业务结构的建立。业务结构中业务的背景、范围和约束条件是其主要需要把控的对象，而业务结构的建立是为了更好的实现业务决策。业务决策过程会产生业务基线和各种变更，同时需要处理业务需求的不同所导致的业务冲突。

在技术管理上，产品化的实施方法包括两个方面。一方面在于确定产品发展策略，产品策略的确定会形成各个产品平台和产品线。而另一方面在于如何推进产品策略的落地，项目化的操作方式以及与产品平台配套的技术平台的确立都属于这一范畴。

关于业务架构和产品化策略的详细内容我们将在第3章中具体介绍。

1.3.2 技术维度

技术维度主要围绕的主题自然是“技术”本身。作为技术管理者，对技术体系的把控也可以分解成三个子维度，包括技术理论子维度、系统架构子维度和技术创新子维度。

1. 技术理论子维度

现代软件开发工具和框架的发展非常迅速，但各个工具和框架背后的理论体系实际上是比较成熟和稳定的。技术理论的切入点比较明确，是对软件开发理论体系的总结和抽象，其目的是用思想和理论去梳理主流技术实现方案的原理。

通过梳理软件体系结构风格、设计模式、架构模式、架构模型等内容，有助于提高技术管理者的技术理论水平，有效指导系统架构的设计和实现。

关于各项技术理论体系的详细内容我们将在第4章中具体介绍。

2. 系统架构子维度

系统架构子维度依赖于技术理论子维度，其切入点在于明确架构设计的层次，本书中的系统架构设计通过业务架构设计和技术架构设计两个层次进行展开。

对于业务架构，实施方式上可以从业务需求出发，针对系统的拆分、集成和扩展三个角度进行设计；而对于技术架构，则更多考虑系统性能、可用性和安全性等非功能性特性。

关于系统架构的层次以及实施方案的细节我们将在第5章中具体介绍。

3. 技术创新子维度

伴随技术变革的基本规律，技术创新需要明确基本的策略和过程。这些策略和过程决定了技术创新是从内部还是从外部进行切入。

对于内部创新，实施上主要包括技术应用和技术演变。前者使用成熟技术解决新问题，而后者偏重于对现有技术进行改造，使其演变成技术问题的解决方案。而对于外部创新，开展合作从外部获取新技术，或者通过应用其他行业成熟技术到现有业务中的跨业创新都是可以使用的创新手段。

关于技术创新的详细内容我们将在第6章中具体介绍。

1.3.3 管理维度

软件开发是一项系统工程，从系统工程角度出发，软件开发可以分成三大部分，即软件实现、项目管理和过程改进。软件实现部分已经在技术维度进行介绍，管理维度则从工程性管理角度出发，包括软件的开发过程和软件经济学等内容，从中我们抽象成项目管理、研发过程和组织管理三个子维度。

1. 项目管理子维度

项目管理的切入点在于从范围、时间、成本等角度出发讨论如何在一定的约束条件下实现系统并完成最终成果的交付。这其中涉及项目管理的通用性知识体系，但也需要根据软件开发的特征进行分析。

对于软件开发而言，需求工程、计划管理、质量管理、风险管理是项目管理重点需要实施和管理的对象。相较其他行业，需求以及系统建模、软件开发范围的分解和工作量评估、技术评审的实施方法以及持续交付思想和工具的应用贯穿整个软件开发的进程。

关于项目管理和系统交付详细内容我们将在第 7 章中具体介绍。

2. 研发过程子维度

软件开发是一系列过程的集合，过程改进围绕这些过程，提出持续优化的方法和实践确保得到令人满意的结果。过程改进的切入点在于通过理解代表性的过程模型，并结合团队目前以及未来的开发状况找到适合自身的过程模型。

研发过程的建设包括过程管理的模型以及研发相关的工程实践，而过程改进同样也有一整套的方法论，无论是传统型的瀑布还是当下流行的敏捷，都崇尚过程改进。而对于特定团队，这些模式和方法都不一定适用，不能照抄照搬，所以研发过程建设的实施方法首先是过程裁剪，通过裁剪建立起符合自身团队发展的轻量级过程模型。

关于研发过程体系建设详细内容我们将在第 8 章中具体介绍。

3. 组织管理子维度

组织管理的切入点在于明确一个组织中需要技术管理者进行管理的视角和边界。对于软件开发而言，向下的团队管理和向外的协商沟通管理是最基本的组织管理视角，但我们也应该注意到向上管理的重要性以及提升管理者本身的自我管理意识。

对于向下管理，实施过程中需要理解技术人员，并通过领导、激励、培训和绩效管理等手段提升团队整个工作效率。对于向上管理，更多则关注结果导向和目标管理。向外管理上，沟通是关键。而对于自我管理，则需要培养个人的管理风格以及处事能力。

关于团队管理以及自我管理详细内容我们将在第 9 章中具体介绍。

1.3.4 维度关系

以上各个维度和子维度虽然各自表现技术管理的某个方面，但也存在依赖关系。图 1-5 描述的是三个维度之前最基本的依赖关系。从图中可以看到行业分析帮助定义产品，技术体系为产品提供实现方法，而管理体系从过程角度为产品开

发提供保障。因此，技术体系和管理体系的建立本质上都是为了实现产品，也就是说技术管理的本质需求是完成产品目标。

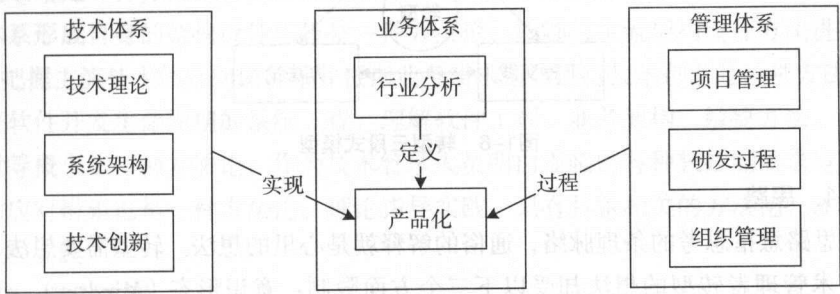


图1-5 技术管理维度之间的关系

不同的行业、不同的业务、不同的系统对于技术管理的维度而言具有不同的展示要求。以互联网行业为例，目前各个领域变化迅速，从行业分析到产品发布的周期也较短，意味着在开发过程管理上适合采用比较轻量级、快速迭代的研发模式，这就需要在项目管理、组织管理上采用与之匹配的模式。另一方面，互联网产品或服务通常面向多个领域，按应用类型区分，通常行业门槛并不高，如果想要快速占据市场，通过技术创新来推动产品化是常见的手段。而面向企业级应用的软件产品中，由于业务复杂且具有一定的行业壁垒，技术更多时候是为了实现业务需求，一个产品的开发周期普遍较长，相应的研发节奏和过程也偏向于采用重量级框架，这些都与互联网产品有较大差别。

1.4 技术开发向技术管理转型

通过前面的章节，我们已经明确了技术管理与技术管理者的基本概念，技术管理的维度也是代表技术管理者从事技术管理活动的主要内容。接下去要解决的问题就是如何从普通的开发人员成功转型成一名技术管理者。

1.4.1 转型成功的三段式模型

转型需要一个过程，任何过程一般都可以抽象成人、工具和流程的组合。但是对于转型过程而言，显然普适意义上的人、工具和流程并不能直接应用。如何找到更加有效的途径来完成从开发人员到技术管理者的转变，本书基于图 1-6 所示的由思路、方法论和工程实践所构成的三段式模型，结合技术管理这一特定转型主题做进一步介绍。

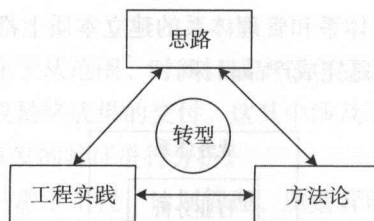


图1-6 转型三段式模型

1. 思路

思路意指思考的条理脉络，通俗的解释就是心里的想法。转型需要想法，但往技术管理者转型的想法却受以下三个方面限制：意识形态（Mindset）、环境（Environment）和决心（Determination）。意识形态是转型的触动点，当我们想去做一件事情而这件事情需要付出很大努力时，通常是意识形态发生了变化，从习惯于根据详细设计文档编写代码并完成功能自测，到根据业务需求抽象出系统模型并转变成架构描述，意识的转变是工作内容转变的前提，意识形态很多时候决定了一个人发展的高度。但一个人所能达到的高度还很大程度受限环境因素，好的环境和不好的环境对个人发展影响巨大，而我们往往无法改变环境，只能适应环境，所以是否具备一个良好的环境也是在转型之前需要进行梳理并做出判断，必要时也应该果断采取行动。思路的最后一点就是决心，当意识形态和环境因素都已经具备，决心变成是否能够转型的关键，毕竟想要成为一名合格甚至优秀的技术管理者可能要比想象的困难。

一般而言，从客观的技术领域进入到以主观为主的管理领域，技术人员会发现这种角色转换要比预想的更具挑战性。甚至许多技术人员对管理者都持一种否定或者观望的态度。他们眼里的管理者似乎是犹豫不决的人，又没有精湛的技术，还常常自以为是。尽管这样，每年还是有许多技术人员接受提拔，进入管理者阶层。这些技术人员相信会找到并解决管理过程中存在的种种问题，正是这种信念促使大多数技术人员接受挑战完成转型过程。然而，并不是所有的技术人员都能获得提拔的机会，对于目前尚未有明确的提升机会但又想往管理者转型的技术人员而言，我们认为思路恰恰是其首先需要考虑的问题。

2. 方法论

所谓方法就是做事的手段、方式、流程，而方法论即一组方法的集合，也就是一组用于确保成功的规则的集合。技术人员想要转型到管理岗位，将要面临一大堆他们不熟悉的问题。对于技术人员，解决技术问题的能力是主要的衡量标准，技术人员自身所具备的方法论也更多地偏向技术体系本身。但对技术管理，技术体系只是一个方面，更多的方法论需要进行理解和掌握。

对技术管理者而言，了解行业发展的大致方向并能从用户研究和体验的角度出发思考并形成自身对业务的理解是一种方法论；能够基于业务需求规划和产品实现策略是一种方法论；了解主流软件架构风格、模式和模型、通过整合各种架构体系形成自身的架构设计思想是一种方法论；能够对主流架构设计方法进行阐述、把握主流技术体系知识领域并根据业务需求适当进行技术创新是一种方法论；围绕软件开发生命周期的系统工程，理解软件工程、业务架构、敏捷方法、产品交付等概念是一种方法论；作为技术管理人员明白面临的各种软技能需求以及相应的应对措施也是一种方法论。理论指导实践，只有具备相关的方法论，才能用于工程实践。

3. 工程实践

在软件开发领域，我们经常提倡使用各种最佳实践（Best Practice）。最佳实践是一个管理学概念，认为存在某种技术、方法、过程、活动或机制可以使生产或管理实践的结果达到最优，并减少出错的可能性。把软件开发的最佳方式和开发人员个人做得最好的事项——总结出来，就是组织的最佳实践。最佳实践包含在技术和非技术领域，包含在对人和事物的处理过程，也包含在技术管理者所应具备的各项软、硬能力中。要想成为一名技术管理者，对业务、技术和管理三个维度以及各个子维度所应该从事的各项活动都应该需要且能够提炼出最佳工程实践作为具体工作展开的输入和模板。

1.4.2 转型思维导图

技术管理者转型面临巨大挑战，挑战来自于技术管理者的工作特性。我们从表 1-3 中进一步看出开发人员与技术管理者之间的区别所在。

表1-3 开发人员与技术管理者的区别

	开发人员	技术管理者
组织中的位置	执行层	规划层
职责范围	技术专项事务	团队
工作对象	事	人和事
工作技能	业务技能	人际和管理技能
评价标准	个人成绩	团队成绩
自我实现	技术专家	管理专家

在完成角色转换的过程中，要避免开发人员与管理人员本身只是层次不同而已的观念。管理也像其他任何一种专业一样，需要专门的知识和才能。在任何一家公司中，管理部门的目标都是完成公司的使命，因此，作为一位想要往管理者

转型的技术人员，首先要明确公司的战略目标。

而当成为一名技术管理者时，会发现你的时间不再属于你自己，你将要花大部分时间与员工沟通交流并指导他们的工作。你已不再像技术开发人员一样有自己的时间坐到办公桌前钻研研究问题。管理部门是公司内部沟通的基础，管理者们构成了公司的信息网络，各种各样的数据通过这个网络流动。每个公司都采用不同的协议来传递这些信息。在你扮演管理者这个新角色时，你应该学会如何包装你的工作，如何滤掉无关的信息，如何辨别哪些是需要你及时处理的哪些又是可以缓一缓或者授权给下属完成的活动。

显然，要做到以上各个方面是困难的，软件行业特定的企业文化以及开发人员特定的思维模式决定了技术管理者不同于一般的企业管理人员。面对技术管理者转型所需要克服的各项挑战，结合转型成功所需要的三段式模型，我们得出了图 1-7 的最高层面的转型思维导图。

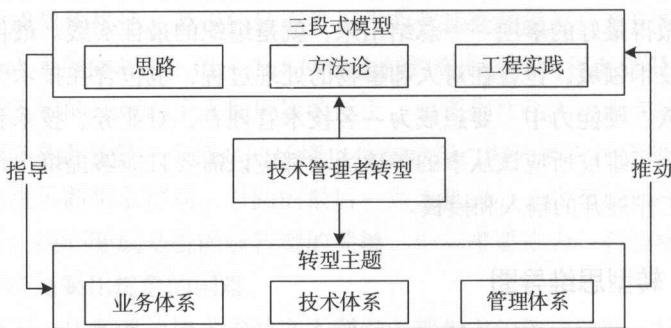


图 1-7 转型思维导图

图 1-7 的上半部分代表包含思路、方法论和工程实践的三段式模型，下半部分代表转型主题，包括业务、技术和管理三大维度。三段式模型指导着转型主题的落实，即对每一个转型主题，思路、方法论和工程实践都是我们进行转型的基本切入点；反过来，转型主题又推动着三段式模型的进一步成熟和改进。该转型思维导图构成了本书的基本行文框架，本书后续章节内容基本按照该图进行展开。

1.5 全书架构与案例

图 1-8 归纳了本书的详细框架，可以看到全书采用类似总分总的结构进行架构组织。第 1 章作为全书的开篇统领全书后续章节，而第 10 章则作为本书的结尾与第 1 章呼应，从“向技术管理者转型”再回归到“成为一名合格的技术管理者”这一话题。结合转型思维导图，其余的第 2 章到第 9 章一共分成三篇，分别对应

业务、技术和管理三个维度，每篇内容再由若干章组成，细分对应各个维度下的子维度。

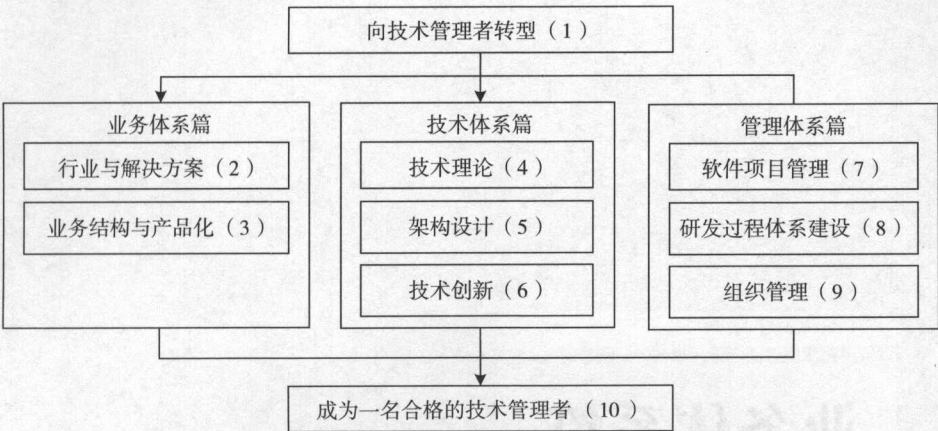


图1-8 全书架构

同时，本书将使用一个具体的移动医疗系统案例来丰满全书内容。移动医疗系统是互联网+背景下传统医疗业务与互联网技术的碰撞，用于帮助患者解决挂号、就诊、支付、查看检查检验报告等场景中的痛点，将部分线下业务转移到线上从而提供更好的用户就医体验。该案例来自于实际产品，我们根据需要做了适当的简化和抽象并贯穿在行业与解决方案、业务结构与产品化、架构设计、技术创新等相关章节中的特定场景中。

1.6 本章小结

本章作为全书的开篇，围绕软件开发人员如何向技术管理者转型展开讨论。转型是一个复杂的过程，需要从意识形态、知识领域、系统工程、软能力等多个方面找到切入点，并付诸工程实践。

技术管理有其固有的特点，本章给出了技术管理的基本定义、演进过程和重要性，同时也对技术管理者这一特定角色做了剖析。通过抽象和提炼，技术管理者的工作表现为几个主要的维度。本章给出了业务维度、技术维度和管理维度等三个主要维度以及各个维度下的子维度。这些维度和子维度构成了本书行文框架和主体内容。

本章最后提炼了转型成功的关键要素。成功转型需要三段式模型，即思路、方法论和工程实践，在业务体系、技术体系和管理体系各个转型主题中运用三段式模型构成了转型的思维导图，为后续内容展开做好铺垫。

业务体系篇

向技术管理者转型

软件开发人员跨越行业、技术、管理的转型思维与实践

本篇共有两章，介绍作为一名技术管理者所需要了解并掌握的针对行业和业务体系的系统知识和相关实践，包括：

1. 行业与解决方案。对业务的理解起源于对行业的理解，技术管理者通过用户研究、用户体验分析、商业模式分析和设计等手段了解和把握行业，并通过解决方案的设计输出对行业的理解。

2. 业务结构与产品化。解决方案是对行业领域的提炼和抽象，其侧重点在于分析现存问题，在解决方案与技术实现之间应该存在一个中间环节，业务结构正是这样一种确保实施效果的中间环节。它描述一种业务需求，这种业务需求可以通过产品化的思维和方式进行抽象和规划。

上述两章构成一种对行业理解的递进关系，从行业分析到解决方案，再从业务结构到产品化，为技术实现提供了各种输入。

所谓行业（Industry）一般是指按生产同类产品或具有相同工艺过程或提供同类劳动服务划分的经济活动类别，如饮食行业、医疗行业、出行行业、金融行业、电商行业等各种具体分类。软件开发本质上也是对各种行业中的特性和行为进行抽象并实现的过程，至于在什么时候开发、开发何种产品、产品本身提供哪些功能等问题都需要对行业进行深入分析之后才能得到合理的答案，这就需要开展行业分析。

当通过行业分析得到产品开发的某个切入点并付诸实施之后，接下来要面对的就是跟行业密切相关的另一个概念，即市场。一个市场规模的大小以及该产品在市场中所能占有的份额决定了该产品的成功与否，这体现的是一种市场成功，但不一定意味着财务成功。市场成功到财务成功需要一个转变过程，这点在当下的互联网行业尤为明显。

任何企业开发软件产品的最终目的是为了**实现财务成功**，一方面软件产品可以通过销售获取收入，而越来越多的软件产品则是通过提供特定的服务实现用户体量到利润的转变。图 2-1 展示了市场成功与财务成功之间的关系。

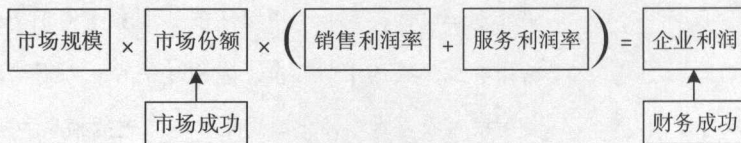


图2-1 市场与财务之间的关系

对于技术管理者而言,并不一定会参与财务相关的营销活动,但也需要通过行业分析了解市场行情和用户行为,进而考虑相应的商业模式。更为重要的是,基于分析所得的结果设计软件产品的解决方案并考虑其中的技术要素是技术管理者的核心工作之一。

2.1 行业分析

本书中的行业分析并不是经济学上的综合应用统计学、计量经济学等分析工具对行业经济的运行状况进行分析的过程，而是从技术管理者的角色和职责出发剖析行业与软件产品开发之间的关系，包括针对用户的研究和用户体验分析，也包括对商业和运营模式的思维方式。

本节对行业分析的讨论本质上是围绕一个话题展开：技术管理者是否需要产品化思维？答案是肯定的。技术思维与产品思维是相辅相成、缺一不可的关系。高超娴熟的技术能力能够支撑产品快速实现和落地。但拥有了产品化思维方式之后，能让技术管理者站在更高的角度去解读产品，避免走弯路。

2.1.1 技术管理者眼中的行业

行业特性在不同人眼里具有不同的表现形式，对于技术管理者而言，自然也有其对行业的理解模型。尤其在当下软件产品处于互联网时代的大背景下，各个行业在自身特性上也表现出一定的共性。

1. 互联网背景下的行业与软件产品特征

我们处在一个互联网空前繁荣的时代，随着“互联网+”概念的提出，越来越多的行业从传统的线下模式转移到了线上模式，意味着软件产品也从以往面向特定用户群体的应用变成面向所有潜在用户的模式。互联网在改变普通老百姓生活方式的同时也给软件产品和软件开发带来了新的特征。

（1）从企业级应用到互联网服务

企业级应用普遍区分行业，各自领域业务背景不一样，并形成了一定的门槛，而互联网应用则可以跨行业，按应用类型区分。从企业级应用到互联网服务，本质上体现的是从信息化到服务化的转变过程。在各个行业都有信息化的需求，而互联网服务能把这些信息集成起来作为一种服务面向特定的垂直领域用户。从这个角度讲，信息只是媒介，互联网起到一种连通作用，而信息与互联网的结合就是服务。

互联网服务较之企业级应用的强大之处在于它的整合作用，服务之间可以通过集成的方式形成一个服务集合，服务集合中的各个服务通过排列组合又可以形成新的服务，并能够把不同行业的服务整合到一起形成一种新的操作方式。图 2-2 展示了基于以上分析的互联网服务基本模型。我们可以看到服务 A、服务 B 和服务 C 通过整合各自行业的信息形成服务化，并可能通过服务之间的整合形成一种崭新的服务 X，这个服务 X 就包含了原先属于不同行业的各种信息。

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

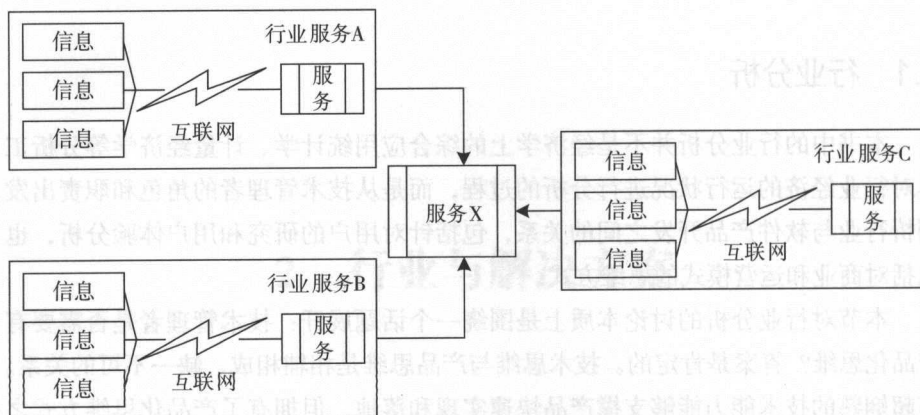


图2-2 互联网服务的基本模型

互联网服务在医疗、教育、饮食、出行等与我们日常生活息息相关的各个行业都形成了一种新颖的用户体验，并逐渐改变了我们的个体行为方式。以医疗健康行业为例，传统的医疗信息化系统面向院内用户，主要包括医护人员和住院患者。而现在市面上各种医疗健康类APP中所提供的服务已远远超出了医护人员和住院患者这些固有群体。基于图2-2中的互联网服务的基本模型，图2-3展示了互联网服务在医疗健康行业的表现方式。

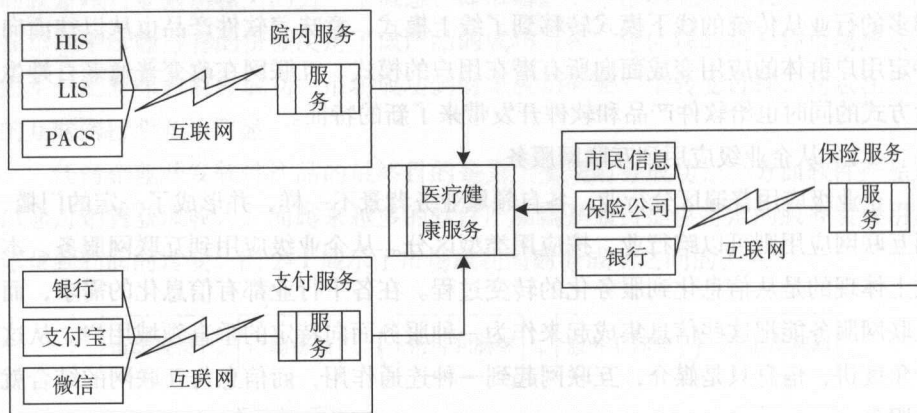


图2-3 互联网服务在医疗健康行业中的体现

从图2-3中，我们可以看到传统的各种院内系统如HIS（Hospital Information System，医院信息系统）、LIS（Laboratory Information Management System，实验室信息管理系统）或PACS（Picture Archiving and Communication Systems，影像归档和通信系统）系统等为互联网化下的用户服务提供了信息来源，这些信息存放在医院内部，但通过互联网的连接作用，用户同样可以在院外环境中使用这些信息。

这是互联网服务的一个重要表现形式，即消除了地域和环境上的限制。

对于医疗健康行业而言，互联网服务的整合作用也在图 2-3 中有所体现。在图 2-3 中，我们又看到用户通过医疗健康类 APP 可以直接进行预约挂号、查看检查检验报告等操作，而这些操作在医院内部一般发生在各个科室或组织。互联网服务能够把这些信息全部整合在一个 APP 中，从而屏蔽了企业或组织上的操作差异性。

另外，我们也可以从图 2-3 中看到更高层次的服务整合。互联网服务不仅仅只是把以前只有在医院里才能实现的操作搬到了互联网上，同时也把原本与医疗健康领域并没有直接关系的其他行业的功能集成到了 APP 中。例如，通过整合支付服务，用户无须在院内的付费窗口排队就可以通过 APP 直接进行在线支付。用户也可以通过 APP 实现诸如申请大病保险等由医疗行为所衍生出来的跨行业的功能，这些功能通过服务化的手段让用户能够切实获取以往所不能获取的体验。

（2）从项目驱动到产品驱动

互联网环境下的软件表现形式从企业级应用转变到了互联网服务，而软件本身的开发模式也从项目驱动转变到了产品运营驱动。

在经典项目管理理论中，范围、时间、成本、质量等各个领域相互关联形成一套完整的体系。对于软件开发过程而言都是以完成一个项目为目标，软件开发的结果通过项目的成功与否来衡量。但在互联网环境下，尽管我们还是会用项目的概念和形式去管理软件开发的过程，但其驱动力已经不是来自于项目章程中的一纸合同，而是来自终端用户的反馈。

随着互联网飞速发展，各种线上产品也越来越多，在竞争日益加剧的现实下，各类互联网产品不得不更加重视用户体验，现在市面上较为成功的产品，往往都是依靠体验而取胜。围绕着如何做出体验更好的产品，整个行业里也形成一定方法论，并诞生了产品经理这样的岗位。用户体验至上的理念在各个行业都表现得非常明显，如在医疗行业中，为了针对“挂号”这一用户需求提供更好的用户体验，各个 APP 在科室如何选择、医生如何确定、挂号费用如何自动结算、号源的自动化分配等各个角度都设计了更为简单、便捷和合理的操作方式。

（3）同质化与运营

经过这些年的持续发展，整个互联网行业的产品能力普遍得到了提升。当下在与普通用户日常生活息息相关的各个行业涌现出了大量互联网服务和应用，随之而来的就是一个明显的问题：同质化。图 2-4 展示了移动医疗掌上医院这个产品 APP 的两个不同服务提供商所开发的系统首页，我们可以看到无论从功能模块还是操作方式都非常相近。不同款 APP 所提供的同一功能给用户带来的体验效果相差无几。



图2-4 掌上医院两款APP的系统首页

想要在产品模式或产品机制上创新，可能会变得越来越难，因为可创新的点可能早就被大家所挖掘和实现。如果你想设计一款能够提供掌上医院服务的移动医疗互联网产品，不管是预约挂号还是在线支付，我们都只要去扒扒市场上已有的类似产品，参考学习下这些产品的设计即可。在这种情况下，很多产品的体验和业务流程，可能都会变得越来越同质化，彼此之前的差异越来越小。于是，决定一个产品是否能够在竞争中脱颖而出的因素，可能就会越来越变成了另一个因素：运营。

产品的体验良好是一个产品成功的必要条件，但成功的产品往往需要跟运营之间保持密切的合作。当下的互联网发展中，运营越发成为市场竞争中的胜负手，运营的策略和方向也会更多影响到产品方面的调整和改动。

2. 技术管理者对行业的理解模型

技术管理者拥有技术背景和能力，但拥有技术背景和能力只是成为技术管理者的第一步。针对某一款产品或一条产品线，拥有一定的行业从业背景，并能对行业的现状和发展有清晰的认识是成为技术管理者的基本要求。业务背景有时候是技术管理人员与技术开发人员之间的一道分水岭，很难想象一个不懂行业的技术人员能够成长为技术管理者。

另一方面，正如前文所述，互联网服务和产品的特点是强调用户体验。如何提高用户体验，就需要研究用户行为找到其痛点，并能够找到提高用户体验的方

法和手段。针对某一个行业，技术管理者需要拥有一定的商业模式设计和分析的能力，这种能力在互联网行业下的表现实际上就是体现在运营上。通过收集和数据分析数据得出产品发展的方向和策略，并付诸商业活动。

业务背景、用户分析和商业模式构成了技术管理者对行业的理解模型（见图2-5），这三者之间存在相关联系。由于各个行业各有特点，需要具体行业具体分析，我们会在全书中穿插移动医疗行业的相关内容。在本节的后续内容，我们将围绕用户分析和商业模式两个方面具体展开。

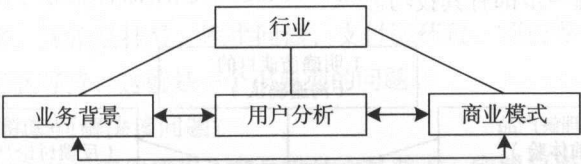


图2-5 技术管理者对行业的理解模型

2.1.2 用户研究和用户体验

1. 研究用户

互联网服务与应用以用户为中心，而用户研究是以用户中心产品设计流程中的第一步。它是一种理解用户，将用户的目标、需求与产品所具有的商业宗旨相匹配的方法。在不了解用户的情况下设计出的产品通常不会符合用户的使用习惯，不能满足用户的体验需求。用户研究的深度和质量直接关系到产品设计的成败。

在互联网领域内，用户研究对于新产品开发和现有产品的优化同样重要。对于新产品而言，用户研究一般用来明确用户需求点，帮助确定产品的设计方向。而对于已经发布的产品来说，用户研究一般用于发现产品问题，帮助优化产品体验。

（1）用户研究的方法

用户研究最基本的实施方法就是用户调研和数据分析。

用户调研从所面对的调研对象的角度而言分为直接调研和间接调研两类。所谓直接调研就是为了产品需要进行有针对性的个别、具体的调研；而间接调研是在目前市面上已经发布的各类资料中获取数据和样本并进行研究，抽取和提炼出所需要的资料。

在调研相关的开展方式上，现场调研是最直接的方法之一。现场调研通过实地收集的信息描述某个特定群体的习惯、想法和行为。例如，如果我们想知道用户在某一家医院中从挂号到医生问诊再到付费取药这个流程中总共所需时间，那

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

么就可以在该医院的实地观察具体某一个患者完成上述所有步骤的总计时间，从而为产品设计提供用户数据。这种方式处于纯粹的观察法，一般在不方便与用户交流或不希望打扰用户的情况下使用。

同样是在现场，用户访谈更为直接和高效，但用户访谈的成本也更高。访谈过程是一个耗用户和客户时间的过程，需要巧妙周全的构建，访谈之前要做好充分的准备，更为重要的是控制访谈的节奏和内容。图 2-6 展示了用户访谈的基本实施框架，可以看到访谈并不只是简单的会面和交谈，而是具有一套流程和方法，流程中的每一步都有其技巧。

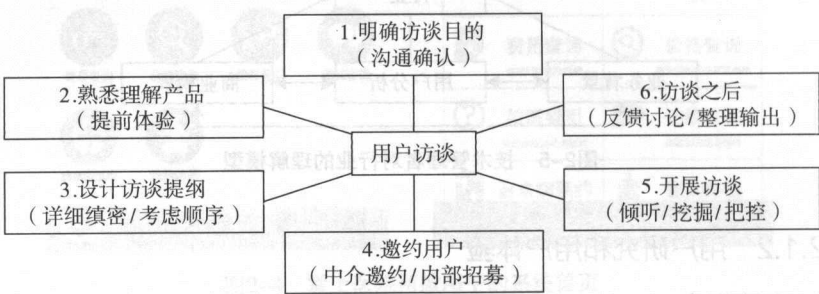


图2-6 用户访谈基本实施框架

现场调研成本较高，且数据采集量不可能很大。可以使用问卷调查作为另一种调研方式。问卷调查假定调研者已经确定所要调研的问题，这些问题被打印在问卷上，编制成书面的问题表格交由调查对象填写，然后收回整理分析从而得出结论。问卷调查覆盖面可以很广，但需要依赖受访者的行为，如果是纸质问卷的话也需要考虑现场操作的成本，而网络问卷调查相对比较容易实施，但不一定能够回收到预计的调研结果。

另一种有效的用户研究方法是焦点小组（Focus Group），通过主持人以一种无结构的自然的形式与一个小组中的被调查者交谈。焦点小组访谈法远不止是一问一答式的面谈，因为在一个群体中，一个人的反应会成为对其他人的刺激，从而可以观察到受试者的相互作用，这种相互作用会产生比同样数量的人作单独陈诉时所能提供的更多信息。在对一系列问题进行展示时，这个主持人是焦点小组发挥作用的关键。通常该主持人需要对产品和业务体系非常熟悉，并具备良好的沟通和引导能力。

(2) 信息分析

通过上述现场调研、用户访谈、问卷调查和焦点小组等手段，我们能够获取不同类型的信息，这些信息可以归为定量信息和定性信息两大类。对于不同的信息类型，所采用的分析方法也会有所不同。

（3）用户痛点

无论使用定量分析还是定性分析，用户研究的最终目的是找到用户的痛点。所谓痛点，一般来说是用户为了更好地生活而碰到的问题。产品就是用来解决用户痛点，痛点很大程度上就是需求，用户分析的结果可以通过痛点来衡量。

如何获取用户的痛点，以对移动医疗系统潜在的用户进行访谈为例，我们可以通过以下三个问题来达到访谈的目的：

- 你正在解决什么问题？

该问题用于收集有效信息作为分析的依据。例如，用户到医院看病，发现医院里面人很多，无论是挂号、医生问诊、支付医药费、领药等各个就医过程中的环节都需要排队等待，这就是一个很典型的问题。

- 目前你如何解决该问题？

该问题用户分析工作流程。产品设计人员并不一定明白用户解决问题的思路和方法，所以需要通过提问来了解各种类型用户的行为方式。例如，在就医过程中，有些人会使用政府公共平台来获取医生的号源，有些人则可能并不了解这些平台。这些都是我们在做产品规划时需要考虑的内容。

- 有什么方法能帮助你做得更好？

这是用户访谈最终的目的，即发现机会，这些机会实际上就是解决用户痛点的潜在方法。例如，用户可能会提出是否可以在去医院之前就进行挂号的预约，也可能认为实现在线支付可以用来解决支付医药费过程中的排队问题。这些答案的梳理和总结就是产品需求最原始的输出。

2. 理解用户体验

根据 ISO 9241-210 标准中的定义^[3]，用户体验（User Experience，UE/UX）是人们针对使用或期望使用的产品、系统或者服务的认知印象和回应，即用户在使用一个产品或系统之前、使用期间和使用之后的全部感受，包括情感、喜好、认知印象、生理心理反应、行为和成就等各个方面。

（1）用户体验的分类

用户体验主要包括三方面内容，即感观体验、交互体验和情感体验。

- 感观体验

呈现给用户视听上的体验，强调舒适性。一般体现在色彩、声音、图像、文字内容、APP 上的功能布局等呈现方式。

- 交互体验

界面给用户使用、交流过程的体验，强调互动、交互特性。交互体验的过程贯穿浏览、点击、输入、输出等过程给用户产生的体验。

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

· 情感体验

给用户心理上的体验，强调心理认可度。用户通过产品能认同、抒发自己的内在情感，那说明用户体验效果较深。情感体验的升华是口碑的传播，形成一种高度的情感认可效应。

(2) 用户体验的表现形式

关于用户体验有很多概念模型，其中几个比较著名的模型包括：

· 5E 模型

5E 模型由 Whitney Quesenbery 提出^[4]，他认为一个用户体验良好的产品应具备有效性（Effective）、效率（Efficient）、吸引的（Engaging）、容错（Error tolerant）和易学（Easy to learn）等五大属性，这些属性的首字母都是 E，所以称为 5E 模型。

有效性是第一个 E，主要表明软件是可用的，帮助用户准确地实现他们的目标。效率是所做工作的速度，可以被量化定义。关于吸引的简单定义就是一个界面的愉快、满意或兴趣程度。容错包含产品防止错误的程度和帮助用户从错误中恢复。易学与产品如何支持初次使用以及更深度的学习相关。显然，这些要素对于某一个产品而言并不是同等重要，例如对于掌上医院类的移动医疗产品，如何能够实现在线的预约挂号背后的有效性是最重要的要素，效率等一般都不会是制约用户不使用该款产品的决定性因素。相反，如果是手术相关的医疗拍摄工具，因为医生在手术过程中时间争分夺秒，效率就变成最重要的要素，完成某项操作点击一次和点击两次就会产生用户体验上的巨大差异。

除了对用户体验各大属性进行了描述，5E 模型还给我们提供了常见的用户体验设计方法，参考表 2-2。

表2-2 5E和可能的设计方法

维度	用户需求	可能的设计方法
有效性	精确性	提供所有关键活动的反馈、消除错误机会、为用户决策提供充足的信息
效率	操作速度	为理想的工作流设计导航，也同时兼容替代方案、提供快捷键、通过交互风格和设计图标提升速度、将界面中的无关元素最小化
吸引	被吸引住	使用清晰的语言和适当的术语、通过适合用户的会话水平设定一个帮助声音、功能结构化以匹配用户任务
容错	有效和确认	将错误转化成替代路径、使用控件有助于准确选择、确信活动容易回溯
易学	及时的信息	通过最少的快捷键和说明使界面有帮助、针对困难或不常用任务创建引导界面

· 用户体验蜂窝

Peter Morville 提出了用户体验蜂窝（User Experience Honeycomb）模型^[5]，他认为产品用户体验的要素有八个方面（见图 2-7），包括 Useful（有用）、Usable（可用）、Desirable（合意）、Findable（可寻）、Accessible（可及）、Credible（信任）和 Valuable（价值）。我们可以看到用户体验蜂窝模型与 5E 模型存在部分类似的要素。

用户体验需要管理。用户体验不是一次简单的行动，它是试图满足需要的一种过程，用户把自己的期望与系统交互生成的结果进行比较，用户体验就是以一种比较的形式呈现出来。用户体验的管理本质上也是一种闭环管理，如图 2-8 所示。

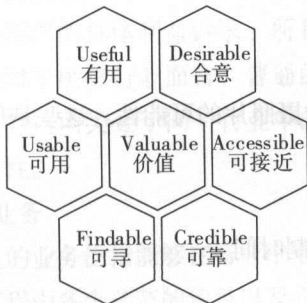


图2-7 用户体验蜂窝模型

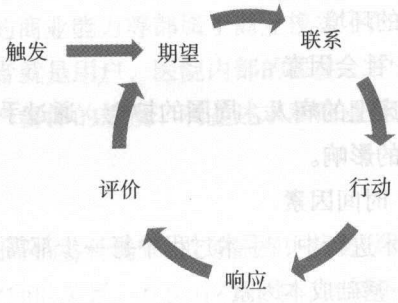


图2-8 用户体验的闭环

用户对于系统的各项操作构成了图 2-8 中用户体验的闭环。首先用户在某些情形触发了某种需要，以及相应的对于满足这种需要的期望。用户期望做什么、期望如何去做、最终期望避免什么等都是用户的需求。有了需求，用户就会尝试使用某款系统，通过判断用户是否在使用系统的核心功能、是否在正确操作等来了解用户需求和系统必要部分之间的联系，并明确系统是否匹配用户的期望。然后，用户就会采取行动并获取响应。通过判断用户期望的行动和实际行动之间有没有不协调，得出对产品的使用评价。基于这种评价，用户将调整他们的期望。如果能够满足期望，用户继续循环的过程，直至最初的需要得到了满足。如果没有满足期望，用户将停止使用系统，并尝试其他的途径，或暂时放弃目标。

（3）用户体验的影响因素

有许多因素可以影响用户使用系统的实际体验。我们对其抽象得到三个核心因素：用户、环境和系统，见图 2-9。针对典型用户群、典型环境情况的研究有助于设

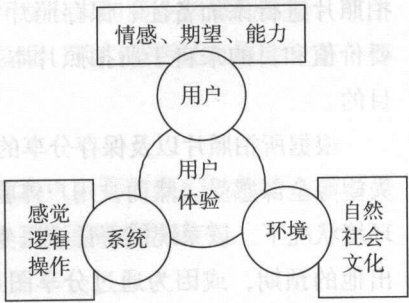


图2-9 用户体验的影响因素

计和改进系统。这样的分类也有助于找到产生某种不良体验的原因。

我们通过一个应用场景来具体分析这些因素对用户体验的影响。在手术室中，医生正在进行某项手术操作，当进行到某一个环节时，该医生希望通过手机能把病人的内部器官状况拍摄下来，添加一下备注之后进行保存或发送给别的医生。这个时候，影响该医生对系统的用户体验主要由两个方面。一方面是医生动机，期望，情绪，认知等精神状态和性格特征，另一方面在于当前的客观资源，即手术室内手术正在进行中。我们先来看一下当前的环境状况：

- 物理因素

手术室专有的空间，各种仪器，灯光，噪音。这些都是这位医生通过感官所感受到的环境。

- 社会因素

病床上的病人，周围的护士，通过手机拍摄照片的可能性。这些是他人对用户体验的影响。

- 时间因素

手术进行中，手术过程中每一步都需要控制时间。

- 基础成本因素

手机网络是否可用，拍摄和转发照片的成本，部分照片是否涉及病人隐私以及可能会涉及的法律约束。

- 任务目的因素

其他正在进行中的活动可能扰乱或中断医生拍照活动，例如病人当前的不良反应，周围护士的协作要求等。

这一特定环境促使医生对拍照具有非常高的操作时效性，即医生希望快速拍照、快速添加备注并进行保存或传播。环境状况正是以这种方式影响了医生与APP之间的相互作用，也就是用户体验。

医生拍照所需要的系统应该包括如下用户界面和功能：拍照的入口、对所拍照片进行添加备注、保存照片、分享照片并获取对方的评价等。该系统的主要价值和目的来自于所拍照片本身，而该系统的所有其他部分都应当支持这一目的。

根据所拍照片以及保存分享的结果，医生的情绪范围可能会经历喜悦、兴奋、失望等全部感受。然而，用户体验的重点在于医生使用系统的感受，在此刻这种环境状况下，该系统能否让该医生以他想要的方式管理图片？该系统是否因为超出他的预期，或因为通过分享图片而获得别人的积极反应而使他感到具有成就感？这些都是用户体验所应该涉及的内容。

2.1.3 商业模式分析与设计

商业模式的分析和设计是从商业维度出发对行业进行剖析以求找到商机和探寻营销手段的过程。这个过程基于前面所述的行业理解和用户研究，侧重于考虑机会和成本。

1. 商业思维

（1）商业模式的关注点

商业模式的讨论通常围绕以下几个点展开：

· 用户

除了我们在上一节中介绍的用户和用户体验，某个特定行业中潜在的用户规模、这些用户的具体利益诉求、所具备的商业能力等都属于商业模式下的考虑范畴。例如对于医疗行业而言，普通的患者就是用户，医院内部的医生、护士也是用户，显然这两类处于同一行业中的用户群体的规模、利益述求和商业能力具有较大差异性。

· 业务

这里的业务泛指能够在行业中形成闭环的各种场景，业务流程中势必涉及用户，也涉及流程中各个环节的角色以及它们之间的关系。在医疗行业中，围绕患者就诊这个业务流程，可以展开的环节就包括门诊挂号、就诊、付费、查看报告等多个业务场景，自然也包含了药物、检查检验报告等媒介以及所有参与其中的人。

· 资源

业务为什么能成为切入点、产品为什么能吸引用户，其背后必定包含了某种稀缺资源，通过获取这种资源，业务才能体现价值，用户也才能获取满意。对于预约挂号等医疗场景而言，医院的号源，尤其是大型三甲医院的号源就是一种稀缺资源，谁能够拥有这些号源，谁就能提供别人无法提供的服务。

· 机会

这里的机会就在于行业目前是处于什么状态，是拥挤不堪的、商战中血流成河的红海，还是通过海阔天空地搞创新、能够开发出未知商业领域的蓝海。对于整个行业而言，我们是处于什么位置、竞争者是谁、我们是否还有机会、这个机会还有多大，这些问题不可避免都需要考虑。

· 成本

如果我们想要踏入某个领域，可能需要资源投入，那么我们就需要考虑需要投入多少成本、一旦投入成本之后产出会有多少、能否回报到成本。

（2）竞争分析

竞争分析是指分析自己的产品在整个行业的竞争格局中处于什么位置，有什

么优势和劣势，从而决定下一步的发展方向。竞争分析的基本工具和手段就是SWOT分析法。

SWOT取自竞争优势(Strengths)、劣势(Weaknesses)、机会(Opportunities)和威胁(Threats)这四个单词的首字母，代表着竞争分析的四个象限，如图2-10所示。而SWOT分析，即基于内外部竞争环境和竞争条件下的态势分析，将与研究对象密切相关的各种主要内部优势、劣势和外部的机会、威胁等，通过调查列举出来，并依照矩阵形式排列，然后用系统分析的思想，把各种因素相互匹配起来加以分析，从中得出一系列相应的结论，而结论通常带有一定的决策性。

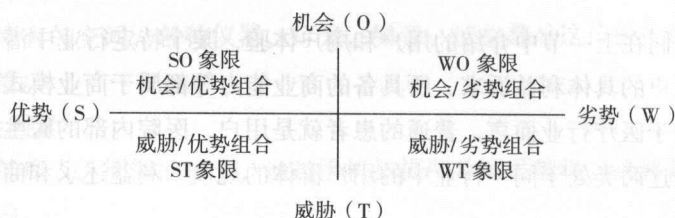


图2-10 SWOT分析法

SWOT分析实质上就是一种盘点思想，对自己的产品和思路进行盘点，对自身的资源进行盘点，对行业中谁能成为朋友、谁是敌人的格局进行盘点。我们参考图2-10中的四象限排列组合，对于分析结果可以做如下规划：

· SO 象限

处于SO象限说明企业拥有强大的内部优势和众多的机会，企业应增加投资、加快软件产品迭代速度，从而提高产品的市场占有率，执行最大限度发展的增长性战略。

· WO 象限

处于WO象限的企业面临外部机会，但自身内部缺乏条件，应采取扭转性战略，改变企业内部的不利条件。开展内部创新活动或从外部获取技术支持是行之有效的策略。

· ST 象限

尽管具有较大的内部优势，但必须面临严峻的外部挑战，应利用企业自身优势，开展多元化经营，避免或降低外部威胁的打击，分散风险，寻找新的发展机会。

· WT 象限

企业既面临外部威胁，自身条件也存在问题，应采取防御性战略，避开威胁，消除劣势。可使用收缩、合并等策略降低企业经营风险。

(3) 行业蓝图

所谓行业蓝图是指对某个行业中各个领域进行细分的结果。一方面细分可以

有不同的层次，另一方面细分的维度包括用户、业务、资源、机会、成本等商业模式所涉及的各个关注点，并给出初步的盈利模式。图 2-11 展示了当下医疗行业的部分行业蓝图。

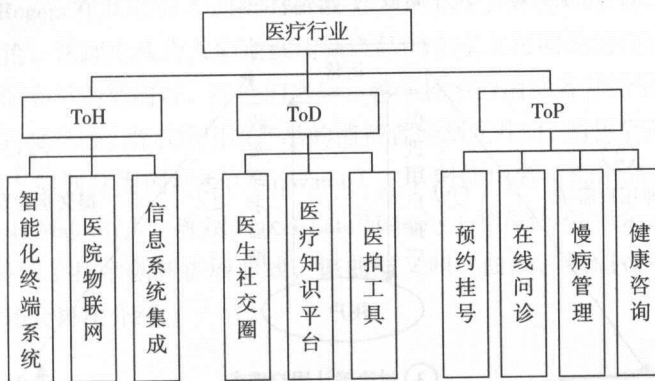


图2-11 医疗行业蓝图示例

在图 2-11 中，我们对整个医疗行业做多层细分：

- 用户层

医疗行业中用户最基本的划分结果为 ToD（To Doctor，面向包括医生在内的医护人员）、ToP（To Patient，面向广大患者）、ToH（To Hospital，面向拥有医疗设备和资源的医院）。

- 服务业务层

对于 ToD 而言，医疗服务主要在医护助手相关的业务，包括在线问诊、医生社交圈、医疗知识平台、医拍工具等。对于 ToP 而言，业务开展方式就很丰富，包括常见的预约挂号、在线问诊等就医类服务，也包括咨询、体检、慢病管理等健康管理类服务。而对于 ToH 而言，完善智能化终端系统、开展院内物联网建设等提高医院管理效率的是其业务需求点。

- 服务形式层

针对服务业务，各种服务可以以不同的形式进行表现。诸如预约挂号、问诊、支付等业务都可以从线下移到线上，以手机 APP 为载体通过互联网服务的形式提供在线服务。而像血压管理等某些健康管理类业务则可能需要智能硬件的支持，另一些保健药品的服务则可以通过电商的形式提供。

- 盈利模式层

对于医疗行业而言，绝大多数服务的盈利模式在于产生交易，通过出售服务的方式获取利润。但对于互联网服务而言，随着用户体量的增大，交易并不一定是获取利润的唯一手段。在用户群体支持下，通过向医院和医生提供医疗器械、药品广告等形式也是一种可行的盈利模式。

2. 运营与数据

互联网行业的岗位笼统的分为技术、产品、运营三大块，图 2-12 展示了这三大块内容之间的关系，我们可以看到开始的第一步和结束的第七步都是运营，即运营是这三者中的推动者，并需要对结果负责。

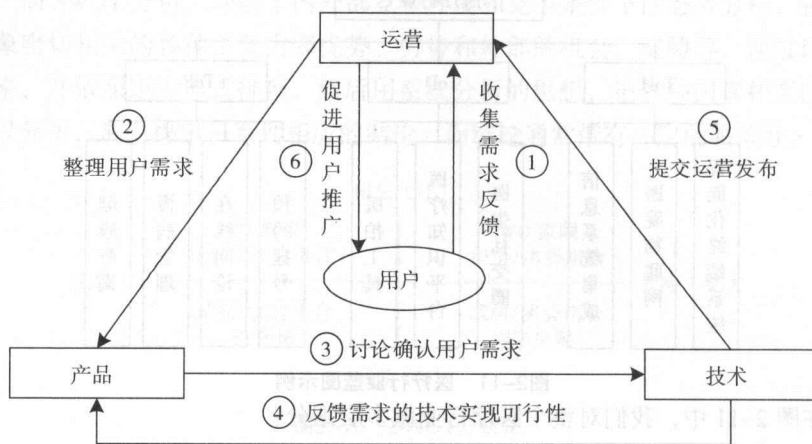


图2-12 技术、产品和运营之间的关系

（1）运营的维度

适合大部分运营工作的三个指标就是拉新、留存和促活，这也构成了运营的三大维度：

· 拉新

所谓拉新，就是为你的产品带来新用户。带来新用户的手段和途径多种多样，可以是策划和制造一个具有传播性的话题和事件，可以是投放广告，可以是在站内做个活动，还可以使用时下流行的微博、微信等工具。在拉新的层面上，一个运营可能会涉及以上各种手段中的一种或多种。

· 留存

所谓留存，就是要通过各种运营手段确保用户被拉到你的产品上之后，最终愿意留下来跟你一起玩。留存所对应的指标叫作留存率，进一步还可以再细分为次日留存、7日留存等。

· 促活

最后是促活，即促进用户活跃，就是让你的用户愿意更频繁、更开心的使用我们的服务。让用户愿意频繁与我们的服务发生连接关系，通过数据分析用户喜好，抓住其痛点并增加黏性，可以采用等级设置、激励体系等手段增加长期活跃性。

（2）运营的活动

运营活动与一个产品的生命周期密切相关，产品生命周期就是产品从进入市

场到退出市场所经历的市场生命循环过程。典型的产品生命周期包括引入期、成长期、成熟期、衰退期等阶段，对于技术、产品和运营而言，意味着技术和产品创新在不同阶段所应该采用的策略也应有所不同。

E.M.Rogers 在其著作《创新的扩散》^[6]中提出了围绕产品生命周期的技术创新扩散理论，该理论认为人们在试用新产品的态度上有明显的差别，每一产品领域都有先驱和早期采用者，在他们之后，越来越多的消费者开始采用该创新产品，产品销售达到高峰，当不采用该产品的消费者所剩无几时，销售额开始降低。同时，该理论把人群划分为创新者（Innovator）、早期使用者（Early Adopters）、早期大众（Early Majority）、晚期大众（Late Majority）和落伍者（Laggards）等五类。图 2-13 展示了技术创新扩展曲线，该曲线又叫 S 曲线，我们在第 6 章中讨论技术创新时还会再次介绍。

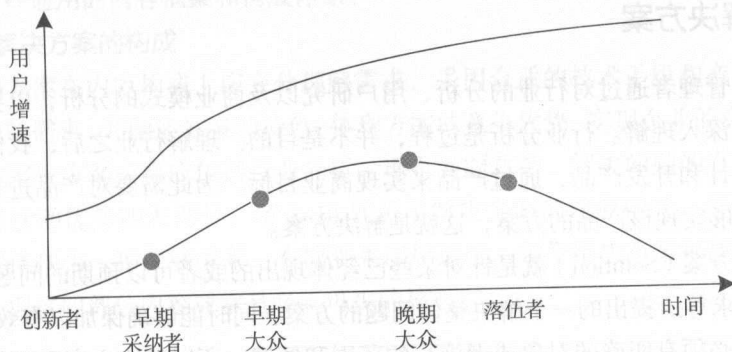


图2-13 技术创新扩散曲线

对于创新者和早期采纳者，在运营模式上应该寻找种子用户，培养产品氛围并确定产品的整体定位。当技术创新扩散曲线逐步上升到顶部，可以规模化增长点，并通过热点营销结合传统媒体渠道加大产品传播力度。而针对晚期用户和落伍者，则应对用户进行活跃度引导，并设法对产品进行变现，并努力孵化新产品。

（3）运营与数据

互联网产品的一大特定就是变化快，为什么要快，是因为有数据在驱动。数据就是运营的武器，通过数据不断获取新需求并调整产品业务功能，从而提高转化率。数据客观反映出一款产品当前的所处阶段，如果产品在某一个阶段表现异常，数据也可以告诉你问题出在哪里。

对于数据分析最基本的两个概念是维度和度量。度量就是具体的数据指标，它通常表现为某个量化的数据值。而维度则是去看待这些指标的不同角度。举例来说，一个面向患者的健康管理 APP 的 DAU（Daily Active User，日活跃用户数量）是一个典型的数据指标，而我们去看待它的时候，可以从日期的维度去看，以便

评估一周或一个月内哪几天流量偏高或偏低，是否存在规律；也可以从 24 小时时间划分的维度去看，以评估每天在不同时间段的流量分布情况；还可以从地域的维度去看，了解不同地区的用户访问使用 APP 的习惯和情况是否存在差异。

运营数据通常以图表等可视化方式进行展现。对于运营数据，一是判断数据是否有一些需要注意的异常情况，如果出现异常数据，一定要分析原因；二是为了给自己的运营工作找到一些方向性的指导。如果在某个时间点 DAU 达到顶峰，那么我们就需要分析是否是开展了某项活动达成了这个结果，如果是，那么后续应该加强这类活动的推广力度。而在另外的时间点上 DAU 数据可能出现明显的下滑后又反弹到正常水平，这就需要我们分析导致这天数据大幅下降的根本原因，以防同样的状况再次发生。

2.2 解决方案

技术管理者通过对行业的分析、用户研究以及商业模式的分析，以实现对该个行业的深入理解。行业分析是过程，并不是目的。理解行业之后，我们的目的是为了设计和开发产品，通过产品来实现商业目标。为此需要对产品进行规划，并给出能够实现该产品的方案，这就是解决方案。

解决方案（Solution）就是针对某些已经体现出的或者可以预期的问题、不足、缺陷、需求等所提出的一个解决整体问题的方案，同时能够确保加以有效的执行。解决方案必须有明确的对象或者施行的范围和领域，所以解决方案和具体行业背景有密切关系。

2.2.1 解决方案设计

解决方案是对一个产品的规划设计，围绕具体的需求而展开，阐述技术、方法、产品和应用。本节中我们将围绕如何设计解决方案这一话题展开，包括基本设计思路、方案的构成以及设计过程中普遍存在的问题。

1. 解决方案设计的基本思路

所谓解决方案，顾名思义就是解决某些问题的方案，所以解决方案中要明确阐述促使产生该解决方案的问题所在。所谓问题，实际上就是对现实的批判和否定，需要用到对比的思路去梳理问题，从存在的问题导出我们的解决方法。问题通常都是综合性的、复杂的，所以需要进行过滤和梳理，重点强调我们的方法中能解决的部分，而不是平铺直叙。问题的引出往往也伴随着问题的背景和解决该问题后的意义，这是第一点。

有了问题，也就有了突破口，接下去推出我们的方案建设内容，也就是具体

展开我们的方案来如何解决问题。对软件行业而言，产品通常就是一套软件系统或者是面向互联网的一种在线服务，可能还附带有一些硬件设备，而产品的建设内容就是产品的功能和结构。系统的业务结构和详细功能展开是方案建设内容的主体，这是第二点。

接下去的第三点，也是最重要的一点就是阐述解决方案的优势，这是解决方案中最需要浓墨重彩的地方，也是整个方案的价值所在。如果一个解决方案不能很好地阐明其优越性，客户如何做出选择这套方案的决定呢？解决方案的优势视具体产品而定，但和处理问题的方式一样，还是可以总结出如安全性、便捷性、人性化、专业性等共性优势，需要我们善于挖掘自身产品的价值点，扬长避短，并以解决方案所带来的效益作为对产品优势的总结。

以上三点构成了设计解决方案的指导思想，解决方案的内容因行业而异，但也存在一些通用的内容框架和构成体系。

2. 解决方案的构成

解决方案在内容构成上应充分理解需求，采用合适的技术手段和产品满足或引导用户的需求，并阐述方案的特色，体现方案的竞争优势，实现企业的商业目标。我们认为解决方案主要应包括行业分析、现有问题总结、解决问题的方法汇总以及产品特性和优势四大部分，图 2-14 展示了解决方案的构成框架，框架中的每个步骤都体现为一种关联关系，有因到果层层递进。每个解决方案可能侧重点有所不同，但这四部分内容通常是必不可少的核心。

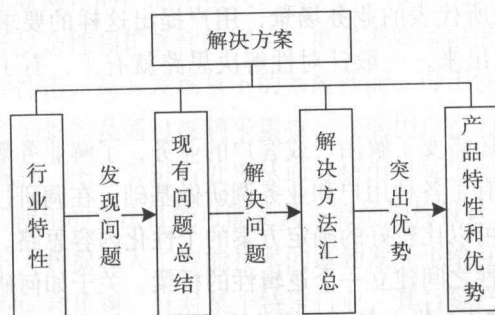


图2-14 解决方案构成框架

图 2-14 中，解决方案具有行业特性，行业分析包括用户研究、用户体验设计、商业模式分析等内容。关于行业分析 2.1 节已经做了详细介绍。现有问题的总结以及解决问题的方法汇总同样来自于用户研究以及对行业内其他竞品的分析。最后的产品特性和优势则是目标产品相较其他产品所特有的内容，也是解决方案的价值所在。解决方案在内容构成上的侧重点即是分析现存问题，提出功能需求及相应技术实现手段，并辅以实施保障，确保用户需求能够实现，并且能比其他同

类产品提供更为优越的用户体验。

除了行业特性、现有问题的总结、解决问题的方法汇总、产品特性和优势等四个主要方面，解决方案中还可能包括系统的实施方案、售后、培训等内容，这些内容大多根据不同的行业特征以及产品需求有所不同，这里不再具体展开。

3. 解决方案设计的最佳实践

解决方案设计中最重要的一点就是结构和思路，有结构就有思路，有思路就有方案。通常对这点进行延伸，我们能梳理和提炼出解决方案设计过程中的最佳实践：

（1）体系

解决方案设计上的问题最主要的就是方案设计者不熟悉自己产品体系所造成，知道一两个甚至更多的产品卖点不难，但难就难在成体系，方案的组织是由成体系的点所构成并形成面，而不是一个个离散的说法。只有当一个人对一个产品思路有体系以后，才能够写出完整的方案。

所以要想写好一个方案，首先要把自己产品的来龙去脉，功能模块，适应领域，典型客户实施情况有一个全面的了解，这样才能建立一个完整的知识体系，然后逐步补充竞争对手知识和一些技术性知识，不断深化自己的知识体系。

（2）思路

设计思路与产品体系密切相关。设计思路如何培养？根本上讲还是要熟悉产品业务，写方案、特别是写针对性方案不仅仅要求了解用户的需求，而且要知道这些需求背后所代表的业务场景，用户提出这样的要求到底想解决什么问题，把这个问题找出来，一般针对性解决思路就有了，有了思路，自然可以很好地写方案。

想要有思路，还需要了解用户或客户的业务，了解业务最有效的方法就是开展用户研究活动，有了各种用户和业务调研做基础，在调研过程中把握用户关注重难点问题，自然可以比较好的确定方案的个性化内容思路。解决方案就是把客户的利益和产品特性之间建立一个逻辑性的桥梁。关于如何研究用户和理解用户体验，我们已经在 2.1.2 节中有所介绍并给出了一些示例。

（3）积累

这一点告诉我们要做积累。一般不经常写方案的人，在写一个方案的时候，即使有想法，有思路，但往往也会很累，就是因为缺少足够的素材。在产品方案设计过程中，关于产品、实现技术等方面的有些内容基本上是通用的，但如果没

有足够积累，每次编制方案就需要花费大量时间去准备，造成方案完成周期过长。

综上所述，解决方案设计上的最佳实践归结于对行业、自身产品的深入了解以及平时对公用资料的积累。

4. 解决方案设计中的技术

解决方案的开发流程如图 2-15 所示，我们可以看到在整个解决方案制作过程中除了业务需求相关的内容之外，还需要完成技术调研和体系架构设计这两个与技术相关的步骤。作为技术管理者，需要全面把控解决方案中的技术部分。

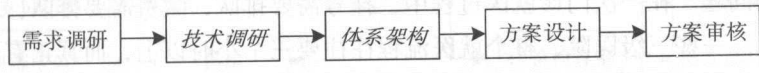


图2-15 解决方案开发法流程

(1) 技术调研

并不是每一种解决方案都需要技术调研，需要技术调研的场景往往涉及技术创新。关于技术创新，本书会在第 6 章中全面展开，这里只需要明确技术调研的目的是确定内部使用或研发何种技术、工具或框架，抑或从外部获取。

技术调研最主要的工作是收集目前市面上的现有产品并通过技术实现方式、架构、功能、扩展性、性能等各个功能性或非功能性需求的对比，从而得出结论。

(2) 体系架构设计

关于通用的软件体系架构设计我们会在本书第 5 章中具体展开，但解决方案中的技术架构显然具备明显的行业特征，各个行业的关注点不尽相同。例如移动医疗行业，由于去医院就诊属于低频事件，系统架构设计并不需要优先考虑高并发等业务场景，而主要实现在现有医院信息化系统的基础上如何快速整合业务和数据从而形成基于互联网化的产品化模式，系统的扩展性是其主要关注点。反观电商行业在考虑业务复杂性和扩展性的同时，更多需要考虑诸如双十一海量用户访问对系统所产生的冲击，高并发场景下的系统性能、可用性是其主要关注点。再如安防行业，业务场景涉及通过视频采集技术完成用户行为监控，并通过算法对这些监控所得的视频、图片等进行处理从而分析出可疑情况，因此海量视频的存储、处理以及背后使用的算法性能可能是其主要的架构设计考虑点。

另一方面，当下大数据和人工智能技术的蓬勃发展也给互联网环境下的系统架构设计带来新的方向和机遇。针对某个特定行业，其目标产品能否充分利用这些最新技术，设计并实现同类产品尚未实现的功能，提供同类产品尚不能提供的用户体验，也是体系架构设计的一个方向。

2.2.2 解决方案示例

本节基于医疗行业中的移动医疗系统，给出设计解决方案的思路、方法和示例。

1. 互联网 + 下的移动医疗

医院门诊服务作为接纳病人的窗口，是病人了解医院、认知医院的重要入口，

也是病人就诊的第一步。门诊服务每天需要接待大量的就医患者，是人群高度集中而流程性较强的场所，也是医院日常管理工作的重要环节。

门诊服务管理工作繁忙、琐碎、工作重复性强而缺乏新意，同时就诊科室种类和医生等信息多而杂，在就诊过程中就医流程冗长、现场排队严重日益成为一个突出的问题。在一次门诊就医过程中，挂号需要排队，缴费需要排队，取报告又要排队。一些三级医院，整个就医流程往往要三个小时以上，而真正有价值的环节，只有不到半个小时。一方面在患者的就医体验上，带来了非常负面的影响，使患者质疑医院的服务质量；另一方面，也需要医院大量的管理和维护投入，加重了就医过程的人力成本和流程成本。如何消除就医过程中不必要的浪费，为患者提供高效和优质的服务是医疗管理者面临的新课题。

随着移动互联网理念和智能手机的普及，为提高患者就医服务质量开辟了新模式。移动医疗系统力图使用创新的理念和技术解决传统门诊就医管理中存在的一系列问题，并创建一种高标准、高质量的就医服务新模式。

通过以上行业分析，移动医疗系统解决方案的模板也就应运而生。该模板在按照 2.2.1 节中解决方案构成所需的行业特性、现有问题的总结和解决方法、产品建设内容、产品特性和优势等内容进行组织。

1. 目标系统应用背景及意义
2. 传统过程中的问题和解决方法
 - 2.1 传统过程中的问题
 - 2.2 目标系统的解决方法
3. 目标系统解决方案的建设内容
 - 3.1 硬件架构设计
 - 3.2 软件架构设计
 - 3.3 目标系统业务功能
 - 3.3.1 各个模块简介
 - 3.3.2 模块详细功能列表
4. 目标系统解决方案的优势
 - 4.1 目标系统的各种优势
 - 4.2 目标系统的效益

2. 移动医疗系统解决方案示例

本节结合上文中的模板对其中部分内容进行展开，以提供解决方案设计过程中的思路和实现方法。

（1）传统过程中的问题和解决方法

传统门诊就医涉及患者和医务人员之间的多次交互，其中常见的交互场景和交互流程如图 2-16 所示。

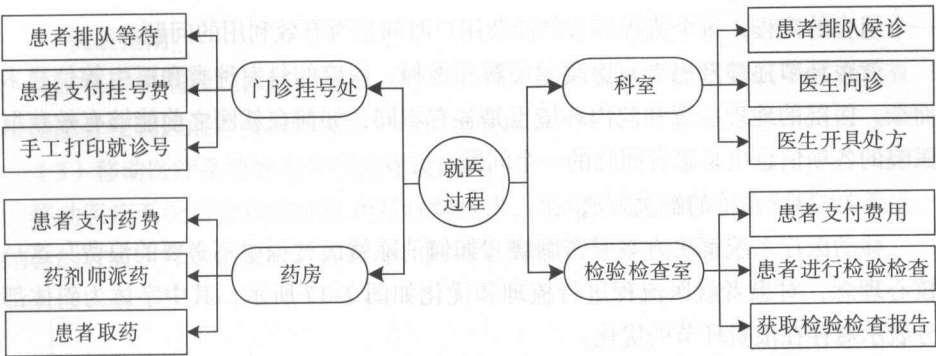


图2-16 传统门诊就医交互流程

围绕“如何消除就医过程中不必要的浪费”这一核心理念，上图中传统模式下的交互流程存在如下问题：

· 就诊排队时间过长

患者一次就诊需要多处交互，典型的交互场景有门诊挂号处、就医科室、药房以及检验检查室等，基于目前国内的就医现状，每个交互场景都可能需要长时间的排队等候。交互场景越多，由于排队等候所造成的浪费也就越多。

· 挂号过程烦琐

对门诊挂号而言，传统流程下患者需要在挂号窗口进行排队等待，通过描述自身症状选择合适的科室并支付挂号费，而医务人员通过与患者确定并打印就诊信息交与患者，这中间的步骤通常比较繁忙、琐碎和重复性强，但又需要医院投入大量成本，患者的就医体验也不理想。

· 候诊时间浪费

对就诊过程而言，传统模式下患者在挂号窗口获取挂号信息后赴某个科室进行就诊，这个过程就需要候诊排队。对于三级医院而言，候诊排队的时候通常比较长，由于患者无法获取对具体就诊时间的预判，所以无法做到候诊时间的合理把控从而造成浪费。

· 取药过程烦琐

对取药过程而言，传统模式下患者首先需要根据医生开具的处方信息进行缴费，然后根据缴费记录到药房才能取药。通过缴费处缴费和药房取药两个子流程才能完成取药，而这两个子流程都有可能需要排队等候，影响用户的取药体验，也加剧院方的管理成本。

· 检验检查报告获取过程烦琐

对检验检查报告而言，一方面患者需要支付检验检查费用进行检验检查，另一方面在检验检查报告出来之后又需要到检验检查科室获取相关报告。这同样是

一个组合的流程，每个流程环节都涉及用户时间是否有效利用的问题。

就医环节还涉及患者对医院的了解和选择，医院的科室种类和医生等信息多而杂，医院的地理位置和院内环境也是各有不同，如何在就医之前能够有效获取医院的各项信息也是患者面临的一个问题。

(2) 目标系统的解决方法

移动医疗系统解决方案紧密围绕“如何消除就医过程中不必要的浪费”这一核心理念，对患者就医流程进行梳理和优化如图 2-17 所示，其中字体为斜体部分表示对存在浪费环节的优化：

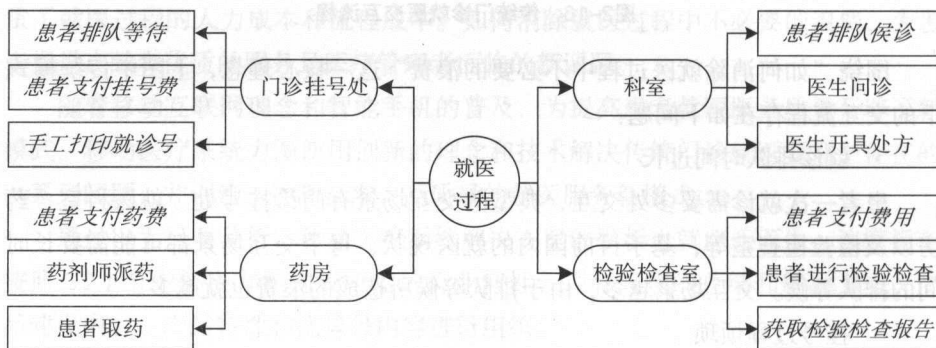


图2-17 门诊就医交互流程优化

移动医疗系统解决方案使用最新的移动互联网技术对门诊挂号、检验检查报告查询、就医指南、支付等环节进行了重新设计和整合，确保就医流程的高效性：

· 门诊挂号

患者通过手机即可完成在线预约并完成挂号费的在线支付，整个过程不需要赴医院现场，也不需要任何排队时间，大大消除了患者在就医过程中的时间浪费。对医院而言，通过自动生成电子化的就诊信息，也在一定程度上缓解了门诊挂号窗口的工作量和成本。同时，根据就诊信息和预约时间段获取的建议候诊时间，患者可以选择在合适的时间进行候诊。

· 检验检查报告查询

患者可以通过手机查看检验检查报告，减少赴医院的次数和时间，及时获取报告信息。报告的电子化实现在降低医院的运营成本的同时，也增强了信息的透明性和同步信息，大大降低了沟通及互动的成本，真正缓解了病人看病难的问题。

· 手机支付

患者使用手机即可完成在线支付，在门诊挂号、药房取药和检验检查过程中都涉及缴费问题，通过手机在线支付可以全部把这三个环节中的支付过程从线下转到线上，极大程度上提高患者缴费的效率，也缓解了医院缴费窗口的工作负荷。

· 就医指南

通过在手机端展现医院的科室、医生、医院地理位置、楼宇导航等信息，有效帮助患者在就医前以及就医过程中的信息导向。

（3）移动医疗系统解决方案的建设内容

移动医疗系统结合移动计算和互联网技术，采用医院前置机、云服务、手机端相结合的方式实行患者就医过程管理。网络架构如下图 2-18 所示：

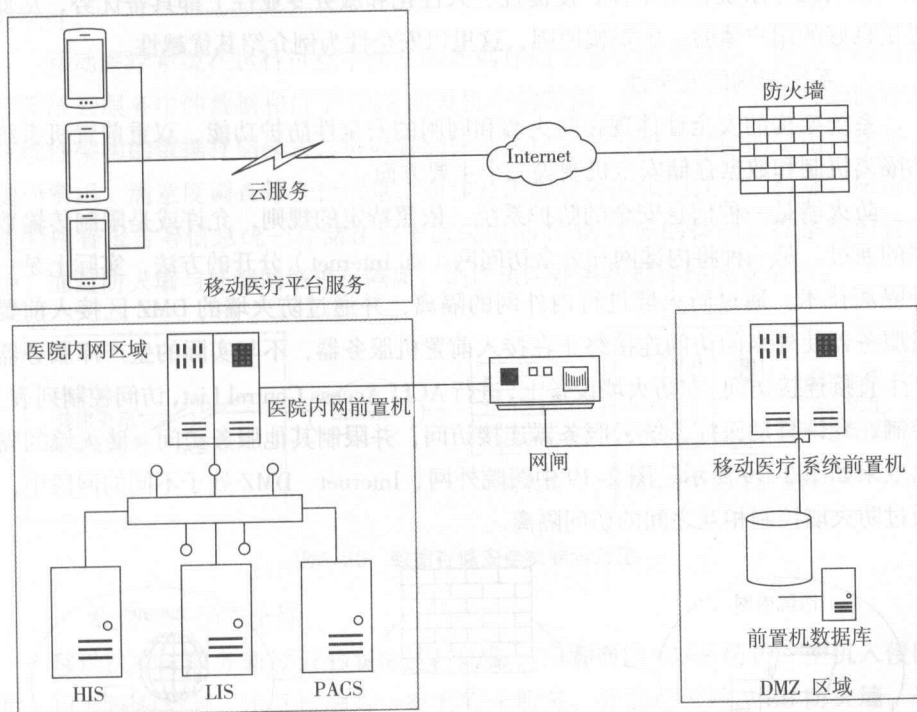


图2-18 移动医疗系统网络架构图

整个流程的技术实现及组网形式分成三大部分：

· 移动医疗系统互联网服务

患者通过安装手机端 APP 并在手机访问移动医疗系统发布的云服务进行服务访问。

· DMZ 区域

DMZ（DeMilitarized Zone）即隔离区，云服务经由医院 DMZ 区域访问位于医院外网环境中的移动医疗系统前置机系统，移动医疗系统前置机系统经由网闸负责与医院内网的各个系统进行交互，同时具备自身的数据存储和处理过程。

· 医院内网区域

医院各个内部系统通过医院内网前置机将基础数据暴露给移动医疗系统前置

机, 医院内网前置机的作用只是数据转发, 确保各个内部系统服务器对外隔离。

移动医疗系统软件架构设计我们会在 6.2.2 节的技术内部创新部分中具体展开, 读者可以先参考相关章节内容。

(4) 移动医疗系统功能

移动医疗系统功能涉及具体业务, 因篇幅原因不展开。

(5) 移动医疗系统解决方案的优势

移动医疗系统在安全性、便捷性、人性化和服务专业性上都具备优势, 从而提供良好的用户体验。因篇幅原因, 这里以安全性为例介绍其优越性。

· 系统架构的安全性

系统架构的安全性体现在防火墙和网闸的安全性防护功能、双重前置机系统的隔离机制和数据存储安全机制等三个主要方面。

防火墙是一种信息安全的防护系统, 依照特定的规则, 允许或是限制传输数据的通过, 是一种将内部网和公众访问网(如 Internet)分开的方法, 实际上是一种隔离技术。通过防火墙进行内外网的隔离, 并通过防火墙的 DMZ 区接入前置机服务器使外来的访问连接终止在接入前置机服务器, 不与实际的生产库服务器产生直接连接访问。在防火墙设备上, 进行 ACL(Access Control List, 访问控制列表)控制, 允许移动医疗系统云服务器连接访问, 并限制其他服务访问。防火墙的隔离效果如图 2-19 所示, 图 2-19 中医院外网、Internet、DMZ 处于不同的网段中, 通过防火墙控制相互之间的访问隔离。

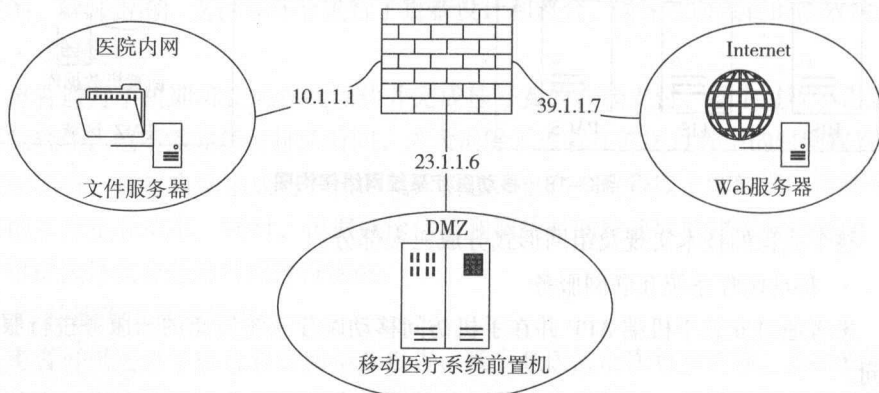


图2-19 防火墙和网闸的安全性

图 2-18 中的网闸也是一种隔离机制, 通过网闸的交换隔离卡技术使内外网完全物理隔离, 通过网闸的专有通信协议保证 DMZ 区的前置机和内网的生产库服务器建立连接, 在网闸设备上, 进行 ACL 控制, 只允许 DMZ 区的前置机和生产库的服务器进行通信, 禁止其他不相关服务进行访问。

移动医疗系统安全体系中包含两台前置机，分别是移动医疗系统自身前置机和医院内网前置机。移动医疗系统自身的前置机位于 DMZ 区域，确保其与 Internet 和医院外网进行隔离；医院内网前置机位于医院内网，是医院内部系统的统一入口，通过网闸和移动医疗系统前置机进行简单的数据转发，避免 HIS、LIS、PACS 等系统直接暴露给移动医疗系统前置机。基于双重前置机机制，通过 Internet 上的移动医疗系统云服务获取医院内网的数据将通过多层隔离确保数据传输的安全性。

移动医疗系统在运行过程中涉及的数据存储主要有两个方面，即位于移动医疗系统云服务中的数据和位于 DMZ 前置机中的数据。图 2-20 是基于移动医疗系统硬件架构的数据存储模式，可以看到移动医疗系统云服务存储的是用户账号、健康资讯、满意度调查等平台信息，而涉及医院业务的预约挂号、科室、医生和检验检查报告等信息统一存储在位于医院内部 DMZ 区域的移动医疗系统前置机中，通过防火墙与外部网络进行隔离，确保医院业务数据存储的安全性。

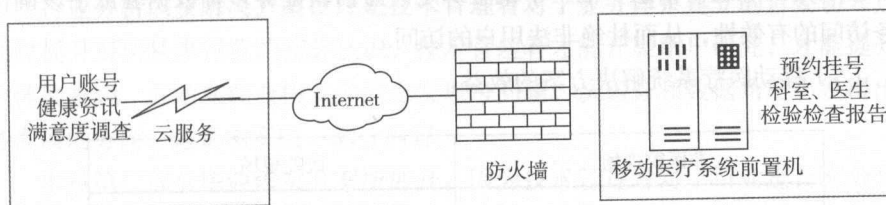


图2-20 数据存储安全机制示意图

· 软件服务的安全性

移动医疗云服务通过云盾系统进行防御，云盾提供 DDoS 防护，主机入侵防护，以及漏洞检测、木马检测等一整套安全服务，并通过云端 WEB 防火墙，进行 ACL 控制，通过这些防御功能保证云服务器的正常运行和安全。

移动医疗系统 APP、云服务和前置机之间都基于 HTTPS 协议对医院内外网数据进行加密传输。HTTPS 即在 HTTP 下加入 SSL (Secure Socket Layer, 安全套接字层)，通过数据加密与传输加密，避免传输过程中数据被篡改和窃取，从而保证患者的信息安全。

移动医疗支付平台采用银行、支付宝、银联和微信通用的数字签名机制，通过分配私钥和 MD5 组成数字签名的技术提供数字安全。

· 业务流程的安全性

移动医疗系统面向广大就医患者，首诊病人在医院现场办理就诊卡，然后进行注册和就诊卡绑定认证即可使用就医服务。在就诊卡绑定认证过程中，患者通过卡号姓名、身份证号、手机号码等在医院现场办卡过程中的预留信息进行身份

认证，只有在本医院现场办卡的患者才能使用服务，确保患者服务访问的安全性。患者的个人账户管理流程如图 2-21 所示。

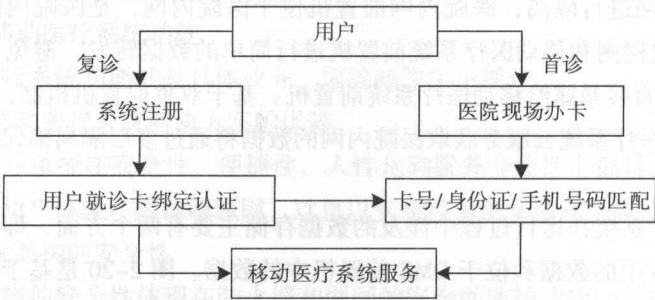


图2-21 患者个人账户管理流程图

对于父母、子女、夫妻等家庭账户而言，同样利用院方预留信息进行验证，同时家庭账户类型权限可做限制，当事人也可随时解除自己的绑定状态。移动医疗系统通过医院遗留信息、唯一手机号绑定、短信验证等多种数据验证手段确保服务访问的有效性，从而杜绝非法用户的访问。

(5) 移动医疗系统解决方案的效益

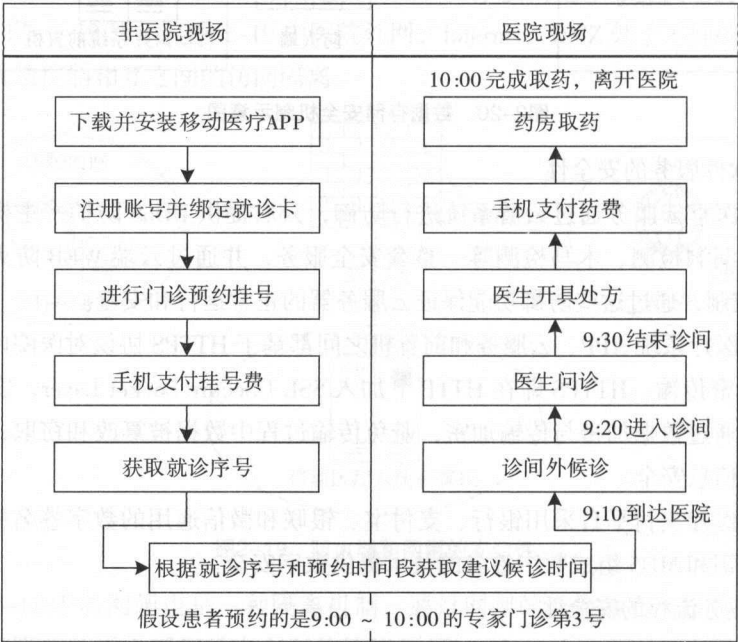


图2-22 患者就医过程对比

通过移动医疗平台的建设，并结合图 2-22 中的流程对患者就医过程进行对比

分析。在通过移动医疗系统 APP 进行预约挂号之后，假设患者预约的是 9:00~10:00 的专家门诊第 3 号。患者就可以根据就诊序号和预约时间段获取建议的候诊时间，在 9:10 左右到达医院直接到对应科室进行诊间外候诊，9:20 进入诊间就诊，9:30 医生问诊结束开具处方，患者花一分钟时间通过手机直接进行缴费然后到药房进行取药，10:00 之前即可完成取药过程并离开医院。整个过程在医院现场停留的时间仅一个小时。

移动医疗平台为患者、为医院的贡献不能简单用数字来定义和表现，但是无形中为患者提供了便利，为医院创造了价值，直接消除了患者在就医过程中的不必要浪费，改善了医院的运营流程，提高了医院的经济效益，进而能够更大程度上满足广大患者对门诊就医的需求。

2.3 本章小结

行业分析以及解决方案设计是技术管理者对于业务体系建立的首要切入点。通过提升对所从事行业的充分认识，技术管理者才能开展用户研究，才能提升用户体验意识，并站在商业模式角度出发思考问题，从运营和数据角度出发得出业务发展的方向。这是本章第一部分内容。

本章第二部分围绕解决方案所展开，因为技术管理者对于行业进行充分分析的目标是能够设计并实现某一款优秀的产品，而在这之前需要给出产品的解决方案。解决方案包括对业务的理解，也包括对技术的预测。本章最后还给出了一个具体示例来展示如何进行解决方案设计。

3 业务结构与产品化

正如上一章中所述，解决方案是对行业领域的提炼和抽象，其侧重点在于分析现存问题，提供功能需求及相应技术实现手段，说明问题能够被解决并强调我们的解决方式上的优势。可以看到解决方案本身并不能作为直接实施的载体，在解决方案与技术实现之间应该存在一个中间环节。

业务结构（Business Structure）正是这样一种确保实施效果的中间环节。它描述一种业务需求，并在此需求的指导下，一个或多个项目交付一个解决方案和符合预期的最终业务成果。

然而，对于复杂的业务结构而言，项目只是一个阶段性的过程，并不代表一个持续化的业务发展方向，产品化才是我们实现解决方案中所阐述的对问题的分析、解决以及提供行业优势的核心手段。产品化的实施包括对产品策略的规划、产品平台和产品线的建立以及产品背后的技术平台的建立。

本章的整体行文思路体现在图 3-1 所示的从解决方案到技术实现的转变关系中，解决方案→业务结构→产品化→技术平台实现了这种转变关系，本章关注于这个转变过程中的中间环节，即业务结构和产品化这两部分内容。

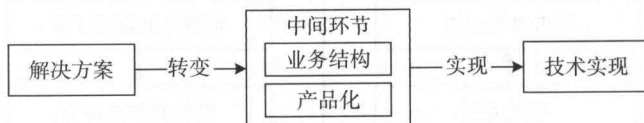


图3-1 从解决方案到技术实现的转变关系

3.1 业务结构

业务结构代表的是需求，定义了解决方案的愿景和实现该解决方案的项目范围。关于业务结构的组成和划分并没有统一的标准，在本书中，我们把业务结构划分成图 3-2 中的表现形式，即业务背景、业务范围和业务约束三方面组成了业务结构的主体。这三个方面在构成整体业务结构的同时，相互之间还可能存在冲

突，因此也需要根据具体的业务结构实现业务决策。

3.1.1 建立业务结构

建立业务结构的过程就是将业务背景、业务范围和业务约束进行拆分并形成可以作为技术实现输入的过程，图 3-2 展示了对业务结构三个方面拆分的一种方式。本节将结合 2.2.2 节中介绍的移动医疗系统的解决方案示例，围绕这些具体的拆分点展开讨论。

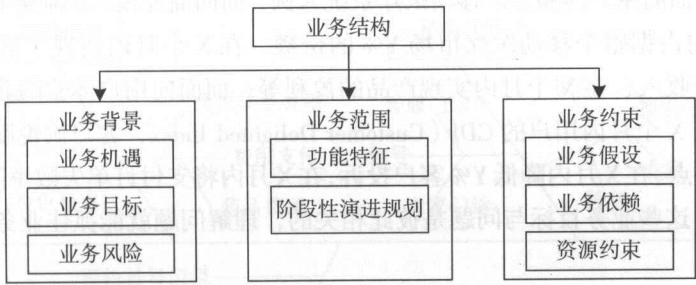


图3-2 业务结构的组成与拆分

1. 业务背景

实现一个解决方案，创造或修改一个产品的初衷往往是为某些人提供有价值的服务和获取一定的回报。业务背景就是要对所面临的机遇、目标进行分析，提出产品发展的愿景并认识到可能所面临的业务风险。

(1) 业务机遇

从解决方案出发，业务机遇一般体现在业务问题的解决或业务流程的改进。对于当下流行的互联网产品或服务而言，业务机遇则更加体现在业务创新。很多业务上的痛点一旦被挖掘，就可以通过技术的应用、产品的研发、商业模式的确立快速形成竞争市场，业务机遇往往与时间因素有直接关系。

通用的说法，业务机遇表现的是满足用户、客户或市场的需要。以医疗行业而言，从以往只能在医院内部进行预约挂号的线下业务模式转变到可以通过手机 APP 进行在线预约挂号的线上业务模式，对于很多传统型医疗信息化企业而言是一个很好的业务机遇，因为这些企业通过现有的各种院内信息化系统已经掌握了大量医院、医生和患者数据，可以通过信息整合快速实现类似预约挂号的互联网服务。但是从业务机遇上讲，这一步的时效性并不长，由于门槛不高且技术实现也都雷同，产品复制的成本就变得很低。另一方面，我们在 2.2.2 节中介绍的移动医疗系统解决的主要用户痛点就是就诊过程中的等待时间，随着医院不断推进院内的信息化建设，如果院内的智能终端系统能够很好地提高就诊过程的效率、

降低患者等待时间的话，这种业务机遇也就随之慢慢消失。

（2）业务目标

业务目标就是用定量的方式计算产品所能带来的商业价值，所以业务目标与2.1.3 节章介绍的商业模式分析和设计密切相关。解决方案中的问题描述了当前业务无法实行目标的各种原因，而这里的业务目标则定义了衡量能够解决或改进问题的各个方面，代表产品的一种成功标准。

在产品设计和实现过程中，往往会存在两种不同的业务目标，一种面向商业模式，一种面向用户体验。以移动医疗系统为例，面向商业模式的业务目标包括：在X个月内占据整个移动医疗市场Y%的份额、在X个月内实现Y的产品销售额以及广告收入、在X个月内实现产品的盈利等；而面向用户体验的业务目标则可以是：在X个月内用户的CDI（Customer Delighted Index，客户愉悦度指数）上升Y个百分点、在X月内降低Y%客户投诉、在X月内将支付订单失败率降低到Y%等。显然，这些业务目标与问题是彼此相关的，理解问题就能抓住业务目标，反之亦然。

（3）业务风险

业务风险用来总结产品业务相关的主要风险，这些风险可以来自于市场竞争、行业背景等外部因素，可以来自于实现过程中的技术问题、对现有业务可能造成的不良影响等内部因素，也可以来自于用户的消费能力、接受能力等因素。业务风险与通常所说的项目风险有一定区别，项目风险更多关注于评估每个风险的发生可能性、潜在影响并采取的任何应对措施。

再以移动医疗系统为例，业务风险也有很多表现形式。国家政策对医疗行业的影响巨大，是否允许将医院内部的信息通过互联网的形式开放给用户是一种风险，因为国家可能会出台政策限制和约束此类行为；另一方面，医院的合作态度是一种风险，因为资源掌握在医院手上，我们需要根据与医院的合作方式来开展我们的工作；技术的实现方式同样存在业务风险，如何保证在互联网环境下的数据安全性，如何确保院内数据访问的高效性等都属于这类风险的范畴。

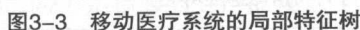
2. 业务范围

软件行业中很多产品都受到一种现象的困扰，即范围的蔓延，产品中不断地添加新功能而导致范围难以控制。范围对解决方案中的概念和应用进行描述，明确实现该解决方案所需要开展的活动。业务范围的确立有助于产品研发的节奏控制和计划安排。

（1）功能特征

业务范围首要的表现形式是功能特征（Feature）。功能特征包括产品的主要特性和面向用户的具体功能列表。功能特征考虑用户如何使用这些功能，一方面

功能特征的梳理视具体行业和业务而定，基本的思路还是分解。关于功能特征的分解，业界也有一些通用的方法有助于更好的表现功能特征，其中比较典型的一种工具是特征树（Feature Tree）。特征树形象展示按逻辑分组的产品特性，并对每种特性进行展开，逐级分解到下一个子特征。图 3-3 展示了移动医疗系统的局部特征树。



对于一个新产品，尤其是互联网产品而言，等所有功能特征全部开发完成再显然不现实的，因为市场和时机变化非常迅速。所以整个产品开发过程体现为一种阶段性的演进规划过程。通过首发版的开发和发布，快速在市场和用户中引爆话题，然后再通过产品的后续版本的迭代和优化，配合运营策略来巩固用户体验。

在整个阶段性演进规划中，对于大型产品而言需要考虑的事情会更多。不同产品需要规划产品线，各个产品之间的共同功能特征需要提炼并整合到产品平台，同时围绕技术体系也需要建立共享的技术平台以提高产品开发效率。关于由产品平台、产品线和技术平台所构成的产品化框架我们将在下一节中具体展开。

3. 业务约束

(1) 业务假设

在产品没有真正面向市场之前，通常需要进行一定的业务假设。业务假设是指在没有百分之百确定信息的前提下先认定决策的正确性。业务假设与业务需求是紧密关联的，如果假设有误，那么业务需求通常也就无法实行，即使实现也可能毫无价值。所以业务假设也是一种约束，会影响到产品的正确性。

以移动医疗系统而言，业务假设也有其特有的表现形式。比较典型的一个假设就是对接医院的数量，因为该产品的用户量来自于医院的就医患者，而患者一般根据其地理位置和就医习惯都有其固定的目标医院。如果系统所引入的医院列表中并没有该医院，那么用户可能就不会使用该产品。假设产品的目标医院数量是 500 家，每家医院每天能吸收 1000 名用户，如果没有达到这个假设，产品目标可能就无法正常实现。

(2) 业务依赖

上面讲到的对接医院数量实际上也是一种业务依赖。移动医疗系统需要提供预约挂号、检查检验报告查询等业务特征，而这些业务特征的实现都需要依赖位于医院内部的数据和系统。当我们去对接医院内部的这些数据和系统时会有很多不确定的因素，包括医院的信息化程度、医院组织方面的态度、接口人的技术和业务水平、具体工作的推行计划和过程等。

业务依赖关系无法满足是导致产品发布产生延期的常见因素。产品中对外部因素的所有依赖都应该明确的记录和管理，除了医院对接，移动医疗系统中还存在对其他第三方供应商交付物的业务依赖。随着产品开发的推进，这些业务依赖关系可能会成为风险，需要定期的监控和调整。

(3) 资源约束

业务约束中还存在着一种客观的、有时候是不可改变的约束，即资源约束。资源约束通常表现为人和物。移动医疗系统中的软件开发人员、产品设计人员、与医院开展系统集成对接的人员等都属于人的资源，而系统部署所需的网络访问能力、数据存储能力、为了提高安全性所需的网关和防火墙等软硬件能力则是部署资源上的约束。

3.1.2 实现业务决策

1. 业务决策的维度

一个产品的业务需求一般具有很多来源，彼此之间可能会产生冲突。业务结构的建立最终需要表现为一种被广泛认可且稳定的形式，但团队各方可能存在利益上的冲突影响到业务结构的稳定性。例如在移动医疗系统开发过程中，对于预

约挂号这个常用功能，系统的开发人员、产品经理和医院相关人员可能会存在如图 3-4 所示的冲突。

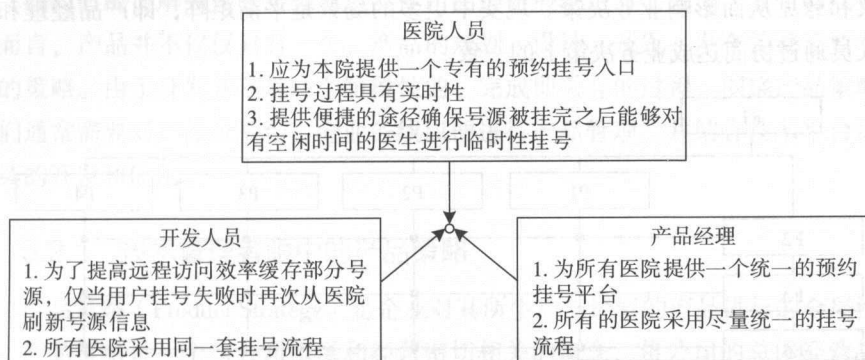


图3-4 预约挂号功能中的业务冲突

显然，我们可以看到图 3-4 中存在一致的地方，但也包含很多相互冲突的业务功能，如产品经理所设想的“为所有医院提供一个统一的预约挂号平台”和医院人员所设想的“应为本院提供一个专有的预约挂号入口”就有根本性的区别。为了解决业务冲突，首先需要明确业务决策的维度。

（1）干系人

所谓干系人（StakeHolder）是指参与到产品规划、设计和实现过程中的个人或组织，它们会根据其利益对结果给予积极或消极的影响。显然，干系人也能影响业务决策，但并不是所有的干系人都对业务决策有显著的影响，干系人管理的首要一点是要识别决策者。围绕干系人的权力和利益，我们需要重点管理权力和利益都与产品有较大关联的干系人。管理关系人的基本法则就一条，即频繁的干系人参与能够减少期望落差。期望落差是指干系人对产品的期望与产品本身所能提供的业务特征不匹配。如果不花精力促使干系人和产品设计开发人员之间达成一致，其后果必然是返工、延期以及降低用户满意度。

（2）组织架构

组织架构对业务决策也有较大影响。在组织结构上，以普遍采用的矩阵式组织（Matrix Organization）架构为例（见图 3-5），它是把按职能划分的部门（图 3-5 中的 F）和按产品划分的小组（图 3-5 中的 P）结合起来组成一个矩阵，管理人员（图 3-5 中的 F1）既同原职能部门保持组织与业务上的联系，又参加产品开发小组的工作。在这样的组织结构下，其业务决策的过程就会跟矩阵的类型有直接关系。

矩阵式组织根据图 3-5 中的管理人员的权利大小，可以分为强矩阵、弱矩阵和平衡矩阵。对于互联网产品开发，这里的管理人员一般就是指产品经理。在强矩阵中，产品经理拥有较大权利，业务决策通常是按照产品经理自身的影响力所

制定；弱矩阵则相反，产品经理权利较小意味着以开发人员为代表的职能团队会有更多话语权，在符合产品整体发展方向的基础上，会提出更多基于技术实现的建议和意见从而影响业务决策。现实中更多的场景是平衡矩阵，即产品经理和开发人员通过协商达成业务决策上的一致。

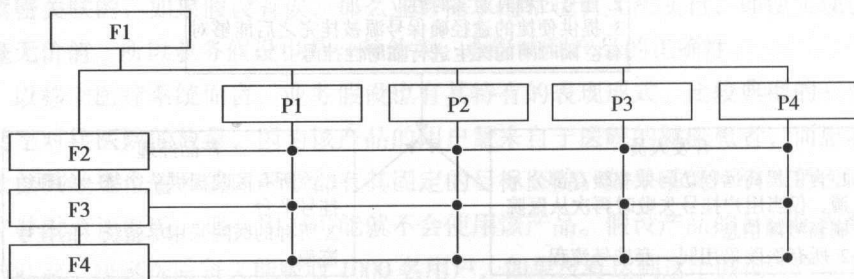


图3-5 矩阵式组织架构

另外，业务决策的制定还涉及产品开发所处的阶段、最高管理层或投资方的期望等因素，同时也受整个组织政治和权利斗争的影响。对技术管理者而言，围绕业务决策所开展的工作在很多时候表现为一种与外部团队进行协商和解决冲突上的能力，我们会在第9章中对如何进行向外管理做进一步展开。

2. 业务基线和变更

当处在某一个特定组织架构中的干系人在特定阶段对业务结构做出了统一的决策，这仅仅是一个开始。产品的开发和演进是一个变化的过程，业务决策同样也需要变化以适应产品的发展目标。因此业务决策表现为从一个统一到另一个统一的过程，如图3-6所示。

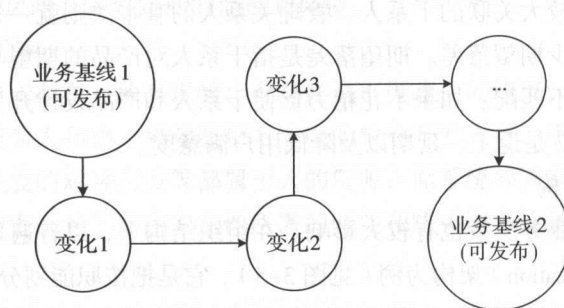


图3-6 业务决策的演变构成

我们把图3-6中统一的业务决策称为业务基线（Baseline）。业务基线是干系人认可的一个业务需求集合，通常作为某一具体的计划发布版本的内容。设立业务基线的目的就在于管理产品的变化性，因为基线是不能改变的，想要改变基线就需要根据业务需求的变更和外部环境的变更修订现有基线并形成新的基线。

3.2 产品化框架

我们已经了解到业务结构通过具体产品进行展现。对于大型、复杂的业务结构而言，产品并不仅仅只有一个，产品的规划、设计、开发、发布等都需要有一定的策略。由于开发并不是一个阶段性的、完成即丢弃的过程，围绕产品策略，我们通常需要对产品分平台、分业务线以便进行产品管理，并结合技术平台进行持续的开发和优化。

3.2.1 技术管理者眼中的产品策略

产品策略（Product Strategy）是企业对其所生产与经营的产品进行的全局性谋划。产品策略是一个与公司市场和经营密切相关的概念，将公司的总体经营战略和产品开发决策联系起来。在本书中，我们对于产品策略的切入点还是站在技术管理者的角度出发，从产品实现以及与技术之间的关系出发来讨论产品策略，并最终给出产品开发框架。

1. 产品策略的基本思想

（1）产品构想

产品策略是从一个清晰的产品构想开始，这是一个明确方向和内容的构想。它描述产品策略希望怎样达到目的以及为什么它能够成功，从而来指导开发一个产品策略的各个要素。一个完整的产品构想必须回答下面三个问题：

- 我们要到哪里去（Where）？
- 我们如何到达那里（How）？
- 我们为什么会成功（Why）？

实际上，我们已经在 3.1.1 节中对如何回答以上三个问题的方法做了阐述，这也印证了产品的构思来自于业务结构，业务结构是产品策略开展的源头。梳理这三个问题的目的在于^[7]：

- 将负责判断新产品机会的人员的努力集中起来

产品构想告诉我们寻找新产品机会的方向。有了产品构想，我们就能从一开始考虑合适的机会；反之，我们就会看到各种关于新产品的想法，而这些想法又可能与公司的优势力量或发展方向相悖。

- 产品构想指导产品开发活动

新产品开发者如果知道公司目标及达到目标的途径，他们的工作会更加成功。这有利于协助他们做出符合战略构想的、设计层上的决策，一个明确的产品构想可使产品开发活动有一个共同的目标。

- 产品构想为技术开发和其他重要的业务部门指明方向

一个明确的产品构想能为技术开发确立总体安排。这也促使技术管理者站在产品策略的高度看待如何开发产品的过程，从而确立产品开发框架。

（2）产品组合理念

从产品开发过程角度出发，产品构想的实现在技术管理者眼中需要两个主要步骤：分解和分层。分解和分层的背后都体现了一种产品开发的核心理念，即组合理念。对于一个大型事物而言，可以通过组合若干小型事物的方式进行构建。分解和分层的目标就是明确产品内部构成的边界，将产品合理划分成一个集合，从而为产品构建提供基础。组合理念表现为图 3-7 中的示意图，我们可以看到通过业务分层、技术平台和业务模块相互组合可以形成产品的基本形态。

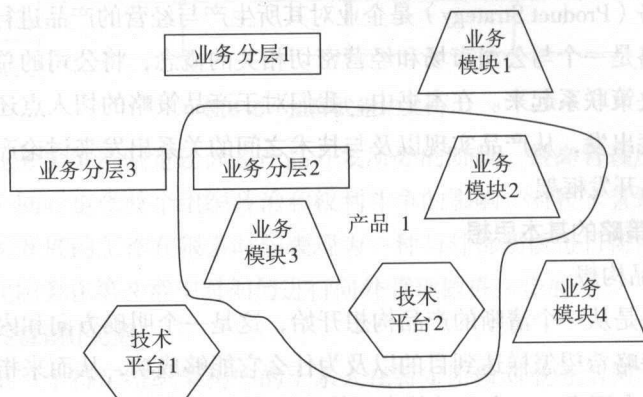


图3-7 组合理念示意图

对于产品开发，分解指的是将产品构想中的业务和技术部分内容想分离。业务和技术分离的优势很明显，因为业务和技术本身就应该独立发展，独立的技术组件能够为产品提供复用支持。而分层则更多关注于业务自身的层次结构，业务也需要区分基础业务和普通业务，基础业务作为业务组件同样为业务组合提供了业务复用性。

2. 产品化框架

基于产品的分解和分层步骤，我们设计了图 3-8 所示的产品开发框架。产品化框架体现的是一种组合理念，并将产品开发分成产品线、产品平台和技术平台三个组成部分。

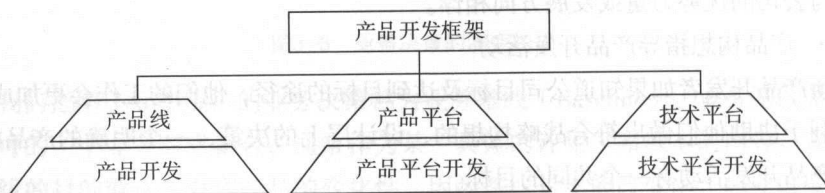


图3-8 产品开发框架

图 3-8 中的产品线偏向于产品线业务计划和项目组合规划及实现，产品平台关注产品生命周期规划和基础架构设计，技术平台则提供产品开发核心技术要素选择和公共组件实现。

3.2.2 产品化框架

在本节中，我们将结合移动医疗系统的产品化进程对图 3-8 中的产品化框架中的各个组成部分做具体展开。

1. 产品平台

对于产品开发而言，并不一定需要建设产品平台。但对于那些采用同一种技术生产多种产品的公司来说，产品平台就显得尤为重要。产品平台决定了成本结构、能力及后续产品的差异。通过将产品平台与产品线区分开来，我们可以更好地将精力集中在产品策略的实现上。

(1) 产品平台的概念

产品平台是共同业务要素的一个集合，指的是一系列产品实施过程中采用的基础业务。从产品的角度看，这些共同业务要素对于最终可发布的目标产品而言没有必要是完整的内容。但这些要素却是目标产品的必要构成部分，图 3-9 展示了产品平台在产品演进过程中的定位。我们可以看到每个产品都会依赖产品平台中一个或多个业务基础组件，而通过在现有产品上添加新的组件就可能构建新的产品，如图 3-9 中的产品 A 和产品 A1。

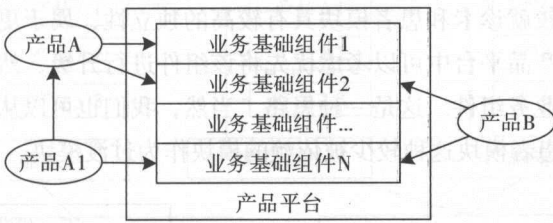


图3-9 产品平台的定位

产品平台包括的内容一方面可以是子系统、模块、组件，这些要素都完全面向业务，通常不能独立提供服务，而需要基于组合理念进行整合应用。在移动医疗系统中，与用户认证、就诊卡管理、病人等内容相关的模块和组件可以放到产品平台中，这些模块和组件自身并不能独立提供面向用户的服务，但在预约挂号、检查检验报告查询过程中都需要使用它们才能形成一个完整的业务闭环。

另一方面，产品平台还可以上升到更高的地位，因为产品线和具体某个产品本身在不断的演进，其所依赖的产品平台也需要随之演进，这种演进往往并不仅仅是个别或一批模块和组件的演进，而是将它们组合起来形成一个具有统一版本

的整体概念。我们需要将现有平台进行演进以形成下一代平台，下一代平台一般会采用一个完全不同的结构来取代原有平台，为防止在下一代平台和现有平台之间出现空当，往往会开发一个过渡平台。如图 3-10 所示，现有产品平台中包含业务基础组件库 A，下一代产品平台中包含业务基础组件库 B，两者之间的过渡产品平台则是对现有产品平台的过渡，包含业务基础组件库 A+。

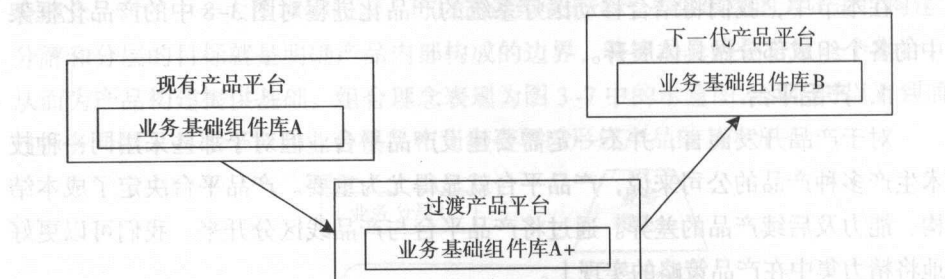


图3-10 产品平台的演进

我们再来看移动医疗系统，用户认证、就诊卡管理、患者等模块和组件在各自独立演进的过程中，还表现出一种整体性。当产品线发展需要这些模块和组件进行根本性的重构时，就会形成图 3-11 中的下一代产品平台。因为用户认证、就诊卡管理、患者等模块和组件之间可能存在较强的耦合，也可能耦合度较低或没有耦合，这样对于过渡平台的设计就可以先替换那些与其他模块耦合度较低的内容即可。图 3-11 给出了一种这些模块和组件可能存在的产品平台演进过程，用户认证模块相较就诊卡和患者模块具有较高的独立性，属于更为基础的业务模块，所以在过渡产品平台中可以考虑优先将该组件进行升级，然后在下一代产品平台中替换所有业务组件。这是一种思路，当然，我们也可以从依赖关系的角度出发，挑选诸如患者模块这种较少被依赖的模块作为过渡模块。

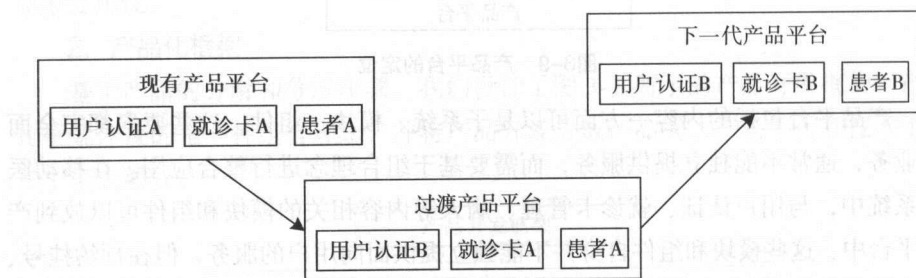


图3-11 移动医疗系统产品平台演进示例

(2) 产品平台设计

产品平台设计的根本要素在于确定产品平台的边界和提供访问产品平台的人

口。产品平台边界的确立在于合理划分模块和组件的粒度，而其所提供的入口则将以平台级别的方式供外部产品或产品线进行业务集成。

一个产品平台的构成要素，包括平台的访问入口以及平台背后的各个模块和组件，这些模块和组件在平台内部存在有序且有限的依赖关系。我们可以看到产品线可以通过产品平台所提供的统一入口以获取产品功能，同时，产品线有时候也可以直接访问产品平台中的某一个共享模块或组件以便整合其功能。作为一项最佳实践，可以将所有的访问之前再加一层门面（Facade）以降低外部使用者与产品平台之间的耦合度。

2. 产品线

对于产品开发而言，一般都会存在产品线（Product Line）的概念。所谓产品线，是指一组相关的产品，这组产品可能功能相似并提供给同一类潜在用户。如果能够确定产品线的最佳长度，就能为企业带来最大的利润，这里产品线的长度是指每一条产品线内的产品数量。

（1）产品线与产品平台

产品线与产品平台的关系如图 3-12 所示，显然产品线由一系列相关的产品所构成，而产品线自身又依赖于特定的产品平台，这三层体现了产品分层思想。同时，我们又可以看出，产品线和产品平台之间并不是一一对应的关系，一个产品线可以依赖于一个或多个产品平台，而多个产品线可以同时依赖于同一个产品平台。

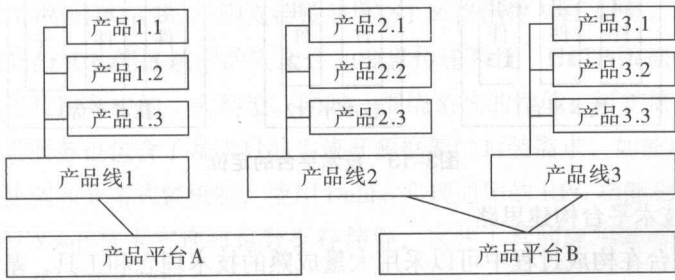


图3-12 产品线与产品平台的关系

（2）产品线设计和实现

产品线的设计和实现因产品而异，很难给出一个统一的标准。以医疗行业而言，移动医疗相关的患者端和医护端 APP 可以作为两个独立的产品，但形成一条产品线，因为这两个 APP 背后的基础数据可能是同一份。但同时，我们可以根据需要把患者端和医护端 APP 拆分成两条产品线，患者端进而演化出面向就医过程的系统和面向健康管理类的系统，从而形成一条独立发展的产品线，医护端也是同样。

3. 技术平台

在图 3-12 中我们看到产品平台 A 和产品平台 B 之间并没有交集，这是一个明显的问题，因为各个产品平台之间势必存在一定的共性。而技术平台构建的目的就是避免出现各产品之间各干各的、很少考虑技术层面重用的现象。从产品设计的角度来说就是缺乏对公共构建模块的规划、开发、应用及维护。我们都建议能够建设独立的技术平台，但许多公司仍倾向于跳过技术平台这个层次而直接进入产品平台甚至产品线开发，技术管理者应该在此过程中起到推动作用。

(1) 技术平台的定位

技术平台的定位非常明确，即为产品平台提供技术支持（见图 3-13）。技术平台的目的在于提供核心技术服务，这些技术服务具有高度复用性和独立性，并实现即插即用的使用效果。技术平台中包含了一系列的技术组件，我们也可以在图 3-13 中进一步明确，产品平台与这些技术组件之间也表现为多对多的对应关系。

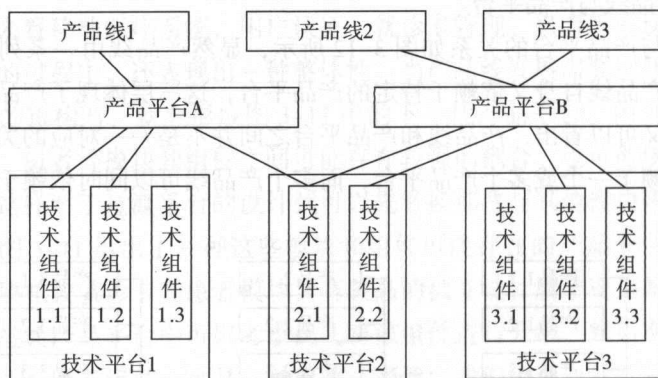


图3-13 技术平台的定位

(2) 技术平台构建思路

技术平台在构成过程中可以采用大量成熟的技术理念和工具，基本思路就是实现服务化。一般认为服务有以下三种主要的表现类型：

· 工具服务

作为应用程序与技术基础设施之间的交叉点，工具服务（Utility Service）的特点是业务领域无关，本质是面向技术、具备高可重用性的低层处理服务，因此能够遵循独立开发和管理生命周期。

工具服务的识别遵循图 3-14 中的模式，以 Java 语言为例，工具服务的识别包括 Java 标准的 API 的封装、公共功能区域的提炼、非功能性需求的抽取以及常见开源框架的应用等四个维度，图 3-14 展示了提取工具服务的这四个常见维度。

因为其业务无关性，相对于其他两种服务，工具服务相对比较容易提炼。

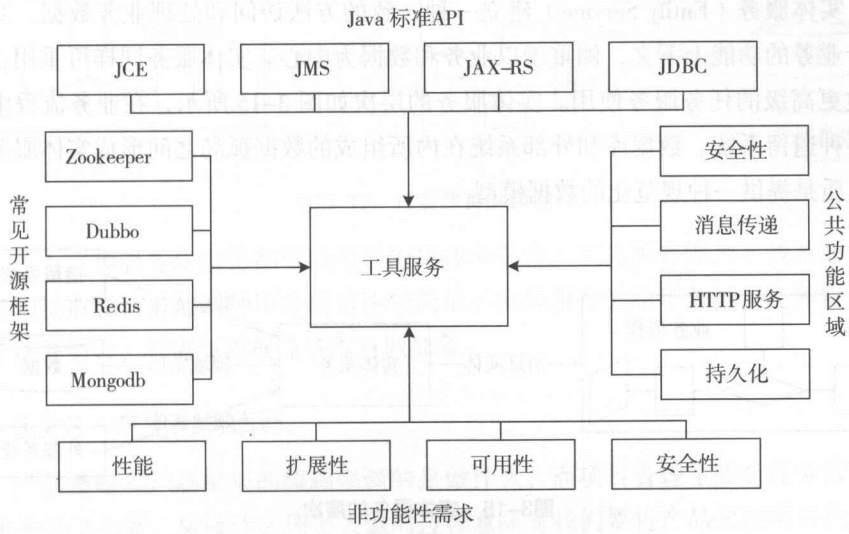


图3-14 工具服务的识别

Java 领域中存在很多开放、强大而常用的 API，如与安全性相关的 JCE（Java Cryptography Extension）提供用于加密、密钥生成和协商以及消息认证码（Message Authentication Code，MAC）算法的框架和实现，JMS（Java Messaing Service）提供面向消息中间件（Message Oriented Middleware，MOM）的 API 规范，JAX-RS（Java API for RESTful Web Services）则支持按照 REST 风格创建 Web 服务。这些 API 可以分别对应到公共功能区域中的安全性、消息传递系统、HTTP 数据传输等各个领域，构成工具服务中的一大类型。同时，围绕系统的性能、可扩展性、可用性等因素，工具服务也包含了基于目前主流开源框架的封装需求，如使用 Zookeeper 实现分布式协调和分布式锁机制，使用 Dubbo 实现通用的 RPC 功能和服务治理，使用 Redis 等 Nosql 技术实现海量数据存储等，这些工具的应用都可以独立于业务并形成统一化、开放式服务。

工具服务也有一些固定的表现形式，包括面向多种应用程序的公共工具服务，如安全性、记录和审核，该类工具服务通常设计成基于 Web 的服务，并开放通用、松散类型接口；封装物理系统资源的资源工具服务，如数据存储 / 消息资源，这类服务处于底层，使用服务门面暴露入口；细粒度并高度特性化的微工具服务，如 XML 加密服务，这类服务通常本地调用，需要考虑性能、无状态性和线程安全，可以作为 JAR 包进行直接引用；面向遗留系统的包装器工具服务，建立标准化服务契约，显然这类服务需要明确所支持的数据和消息模型。

· 实体服务

实体服务（Entity Service）建立一种一致的方法访问和处理业务数据，对应基于业务的功能上下文，侧重于以业务和数据为中心。实体服务同样可重用，一般被更高级的任务服务使用。实体服务的层次如图 3-15 所示，在业务流程中包括各种遗留系统、数据库和外部系统在内所组成的数据孤岛之间形成实体服务，其本质是提供一种规范化的数据模型。

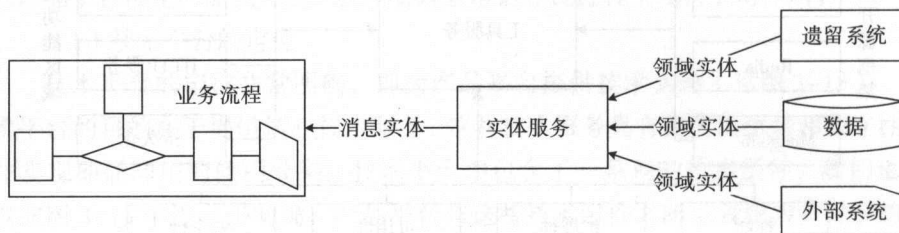


图3-15 实体服务的层次

实体服务中包括领域实体（Domain Entity）和消息实体（Message Entity）两种实体，领域实体是指服务库中规范化的数据模型，而消息实体特定于实体服务契约的数据模型，这两种实体体现在图 3-15 中就是实体服务的上下两端。以常见的客户（Customer）这个概念而言，领域实体中可以包括账户（Account）、订单（Order）和地址（Address）等信息，但当实体服务暴露对外的接口时可能并不一定需要所有的领域数据，消息实体作为领域实体的包装器和简单版本，就能作为一种灵活的、与领域实体相互独立的数据传输和转换机制，领域实体和消息实体示例见图 3-16。

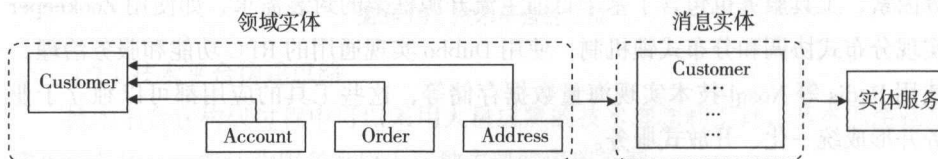


图3-16 领域实体和消息实体示例

· 任务服务

任务服务（Task Service）关注实现业务相关逻辑，很大程度上由组合逻辑组成，通常需要维护状态。任务服务在靠近组合服务的位置承载，体现为一种组合结构（见图 3-17）。在任务服务中，确保使用实体服务限制返回数据量，服务调用携带消费者信息和上下文并决定事务边界。

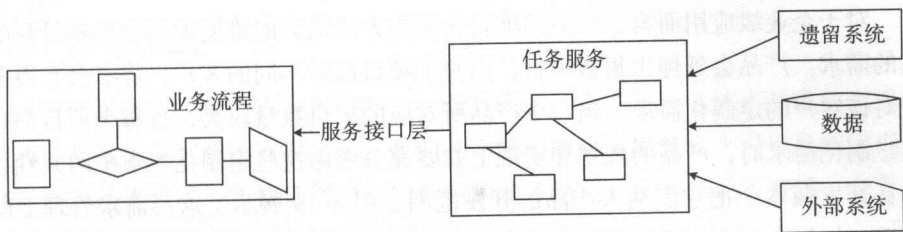


图3-17 任务服务的组合

以上三种服务化思想都可以用于构建技术平台，工具服务因为不涉及业务，识别和提取相较另外两种服务而言比较简单。实体服务关注于数据，任务服务关注于业务组合，都需要根据业务本身做抽象。

3.2.3 产品化与项目

产品策略和产品框架的梳理强调的是做什么，而项目管理主要是任务管理，侧重的是怎么做。从做什么到怎么做的过程意味着我们要将产品通过项目化的方式进行实现。

1. 产品与项目的关系

项目的目标是在规定的时间内，利用有限的资源，高质量地完成某个特定用户的需求。而产品的目标是解决问题，或者说满足一些用户的通用需求。产品不存在完成的说法，因为产品是不断更新的，直到被新产品替代生存周期才结束。一个产品的生命周期可能会由多个项目阶段组成，产品与项目的这层关系如图3-18所示。

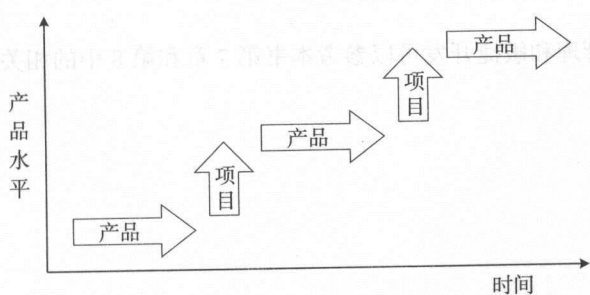


图3-18 产品与项目的关系

2. 项目化实施方法

图3-18中我们可以看到产品管理中包含项目管理，从中得到的启示就是可以把产品的每一个版本的开发过程用项目管理的方式进行管理。不同类型的软件产品其项目化实施方法也有所不同。

对于企业级应用而言，产品的项目化实施方法最大的难度在于管理项目与产品的需求。产品会延伸出很多项目，而每个项目面对不同的客户，势必会有很多针对该客户的定制化需求。当一个产品所对应的项目数量较大，且每个项目都存在定制化需求时，产品的规划和实现上就要充分考虑这些定制化需求中的共性，并在开发版本中把它们纳入产品的开发范围。图 3-19 展示了这层需求管理上的关系，如何合理的分析、划分和整合这些共性需求成为企业级应用产品开发过程中的最大难点。

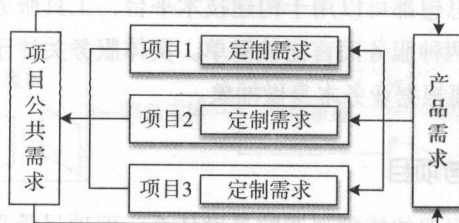


图3-19 项目与产品的需求管理

对于互联网产品而言，产品的主导作用更为明显，因为产品通常面向的就是互联网上的广大潜在用户，而并不是某一个客户，意味着很少会有客户提出的定制化需求，产品需要提供的是面向所有用户的统一的用户体验。互联网产品的这种特性就弱化了项目的概念，以至于通常使用基于敏捷思想的迭代（Iteration）而不是项目的方式在推动产品的演进。例如在基于 Scrum 的产品迭代开发模型中，每个迭代以一种被称为 Sprint 的形式管理产品开发过程。相较企业级应用，互联网产品迭代相对比较简单，不需要过多考虑外部的输入，只要围绕产品本身展开规划即可。

关于项目管理和敏捷开发可以参考本书第 7 章和第 8 中的相关内容。

3.3 本章小结

紧密承接上一章内容，本章和上一章构成了本书中对业务体系的理解和建立过程。当我们得到解决方案之后，我们并不能将解决方案转化为可以直接实现的产品，而是需要进一步梳理解决方案背后的业务结构。本章从业务背景、业务范围和业务约束等三方面对业务结构进行了展开介绍。

有了业务结构，我们就能设计产品化框架。产品化框架设计之前需要明确产品策略。有了产品策略，我们把产品化框架拆分成产品平台、产品线和技术平台三个主要方面。最后，产品化与项目息息相关，本章最后也讨论了产品与项目之间的关系以及对产品进行项目化的实施方案。

技术体系篇

图3-19 项目技术体系图

对于互联网产品而言，产品的主体是用户界面，因为产品是面向终端用户的，而终端用户是互联网产品的核心。因此，互联网产品的技术体系应该围绕用户界面展开。图3-19展示了项目技术体系图，该图是一个树状图，展示了项目的技术体系结构。该图分为三个主要部分：前端、后端和数据库。前端部分包括HTML5、CSS3和JavaScript。后端部分包括Java、Python和PHP。数据库部分包括MySQL、Oracle和MongoDB。此外，该图还展示了中间件、运维和测试等技术体系。该图是一个树状图，展示了项目的技术体系结构。该图分为三个主要部分：前端、后端和数据库。前端部分包括HTML5、CSS3和JavaScript。后端部分包括Java、Python和PHP。数据库部分包括MySQL、Oracle和MongoDB。此外，该图还展示了中间件、运维和测试等技术体系。

关于项目技术体系图，可以参考图3-19所示的技术体系图。

向技术管理者转型

软件开发人员跨越行业、技术、管理的转型思维与实践

本篇共有三章，介绍作为一名技术管理者所应掌握的各项技术技能，包括：

1. 技术理论。理论指导实践，原理和理念一方面能够指导日常的系统设计工作，另一方面也是熟悉并掌握目前技术主流工具和框架的基础。软件设计的原则、架构风格、设计模式、架构模式和架构模型构成了完整的技术理论体系。

2. 架构设计。架构设计在层次上可以分为业务架构和技术架构。系统拆分、系统集成和系统扩展构成了基本的业务架构模型，而系统性能、系统可用和系统安全则更偏向于技术架构。

3. 技术创新。互联网时代对技术管理者的技术创新能力提出了更高要求，技术创新从实现策略上可以分为内部创新和外部创新。内部创新又可以进一步采用技术自身开发、技术应用和技术演变等创新模式；而对于外部创新而言，外部获取和跨业创新是比较典型的实现手段。

对于技术管理者而言，掌握全面的技术体系是工作开展的基础。在借助技术力量为产品化提供支撑作用的同时，技术创新能够反哺业务，确保产品的市场竞争优势。

4 技术理论

有了前面两章介绍的业务和产品体系，我们就可以开展技术体系的建设。在本书中，围绕软件开发理论体系，我们从两个方面切入：

- 软件设计的基本原则有哪些？
- 这些基本原则的表现形式又有哪些？

通过对这两个问题展开讨论，我们得到的是可以普遍应用于各种应用系统、基础框架以及复杂场景构建的基本原理和设计理念，这些原理和理念一方面能够指导我们日常的系统设计工作，另一方面也是熟悉并掌握目前技术主流工具和框架的基础。

任何一种软件，包括从底层的操作系统到运行在操作系统之上的工具框架再到使用这些工具框架开发出来的各种业务应用系统，每一种软件系统在开发过程中，每一步都会遵循一定的设计原则，都可以抽象出固定的风格和模式。这些设计原则、风格、模式是我们在日常工作中应对技术困难和风险时的强大武器，需要技术管理者理解、掌握并能进行灵活应用。

4.1 软件开发理论体系

所谓理论指导实践，软件设计活动首当其冲需要理解并掌握的一些设计原则，包括面向对象的设计原则、面向组件的设计原则以及其他很常见但又不容易归类的各种原则。本节对软件开发理论体系的讨论从这些设计原则切入，并引出技术理论的具体表现形式。

4.1.1 软件设计原则

1. 面向对象原则

(1) 开闭原则

所谓开闭原则（Open Closed Principle，OCP）指的就是“软件实体应当对扩展开放，对修改关闭”，简单讲就是软件系统中包含的各种组件应该在不修改现

有代码的基础上引入新功能。开闭原则中“开”，是指对于组件功能的扩展是开放的，是允许对其进行功能扩展的；开闭原则中“闭”，是指对于原有代码的修改是封闭的，即不应该修改原有的代码。

（2）里氏替换原则

里氏替换原则（Liskov Substitution Principle, LSP）中说，任何基类可以出现的地方，子类一定可以出现。里氏替换原则可以理解为是对开闭原则的补充。实现开闭原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏替换原则是对实现抽象化的具体步骤的规范。

（3）依赖倒置原则

依赖倒置原则（Dependence Inversion Principle, DIP）就是说要依赖于抽象，不要依赖于实现。也即意味着要针对接口编程，不要针对实现编程，具体表现上应当使用接口和抽象类进行各种类型声明以及数据类型的转换。可以认为开闭原则与依赖倒置原则是目标和手段的关系。如果说开闭原则是目标，依赖倒置原则是到达开闭原则的手段。如果要达到最好的开闭效果，就要尽量的遵守依赖倒置原则。

（4）单一职责原则

单一职责原则（Simple Responsibility Principle, SRP）强调一个类应该只有一个职责，并把职责定义为变化的原因。每一个职责都是变化的一个维度，如果一个类有一个以上的职责，这些职责就耦合在了一起，当一个职责发生变化时，可能会影响其他的职责，这就会导致脆弱的设计。另外，多个职责耦合在一起也会影响复用性。如果发现一个类有多于一个的职责，就应该通过分离接口等方式做到尽量解耦。

（5）接口隔离原则

关于隔离的含义就在于类之间的依赖关系应该建立在最小的接口上，一个类不应该依赖它不需用的接口。接口隔离原则（Interface Segregation Principle, ISP）就是说建立单一接口，不要建立臃肿庞大的接口。接口隔离原则与单一职责的审视角度是不相同的，单一职责要求的是类和接口职责单一，注重的是职责，这是业务逻辑上的划分，而接口隔离原则要求接口尽量细化，接口的方法尽量少。

以下五条原则一般被称为面向对象领域的S(SRP).O(OCP).L(LSP).I(ISP).D(DIP)原则。

2. 组件设计原则

组件设计原则有时候也称为分包（Package）原则。任何一个软件系统都可以看作是一系列组件的集合，良好的设计能够把系统分解为一些大小恰到好处到好处的组件，从而使每个开发团队都可以只关注单个组件而无须关心整个系统。对于组件而

言，最核心的关注点就是内聚（Cohesion）和耦合（Coupling），所谓内聚是指一个组件内各个元素彼此结合的紧密程度，而耦合指的是一个组件内不同模块之间互连程度的度量。基于这两个关注点，组件设计原则也包括组件内聚原则（Component Cohesion Principle）和组件耦合原则（Component Coupling Principle）两大类。

组件内聚原则包括：

（1）重用 - 发布等价原则

重用 - 发布等价原则（Release-Reuse Equivalency Principle, REP）关注粒度，强调重用的粒度等于发布的粒度。重用 - 发布等价思想从用户观点的角度上为我们规范了组件设计的原则：在设计组件时，组件中应该包含的元素要么都可以重用，要么都不可以重用。

（2）共同封闭原则

共同封闭原则（Common Closure Principle, CCP）关注变化，即一个组件不应该包括多个引起变化的原因。组件中所有类对同一种性质的变化是共同封闭的，一个变化如果对一个封闭的组件产生影响，则对该组件中的所有类产生影响，但对其他组件将不产生影响。该原则类似开放封闭原则，即对修改应该是封闭的，但对扩展应该是开放的。从这个角度看，组件越大越能满足共同封闭原则。

（3）共同重用原则

共同重用原则（Common Reuse Principle, CRP）关注重用，认为一个组件中的所有类应该是共同重用的，如果重用了组件中的一个类就应该重用组件中的所有类。即放入一个组件中的类不可分开，仅仅依赖其中一部分类的情况不应该存在。显然，根据共同重用原则，组件应该越小越好。

可以明显看到共同封闭原则和共同重用原则具有互斥性，不同的原则面向不同的场景和生命周期。而组件耦合原则也包含以下三条设计原则：

（4）无环依赖原则

无环依赖原则（Acyclic Dependencies Principle, ADP）认为在组件之间不应该存在循环依赖关系。通过将系统划分为不同的可发布组件，对某一个组件的修改所产生的影响不应该扩展到其他组件。

（5）稳定抽象原则

稳定抽象原则（Stable Abstractions Principle, SAP）认为组件的抽象程度应该与其稳定程度保持一致。即一个稳定的组件应该也是抽象的，这样该组件的稳定性就不会无法扩展。另一方面，一个不稳定的组件应该是具体的，因为他的不稳定性使其内部代码更易于修改。

（6）稳定依赖原则

稳定依赖原则（Stable Dependencies Principle, SDP）认为被依赖者应该比依

赖者更稳定。一个好的设计中的组件之间的依赖应该朝着稳定的方向进行。一个组件只应该依赖那些比自己更稳定的组件。在图4-1中,我们认为组件X是稳定的,因为X被很多其他组件依赖,相当于责任担当着。而X没有依赖别的包,所有它具备很高的独立性。同时,我们认为组件Y是不稳定的,因为Y没有被其他的组件所依赖,但Y自身依赖很多别的组件。

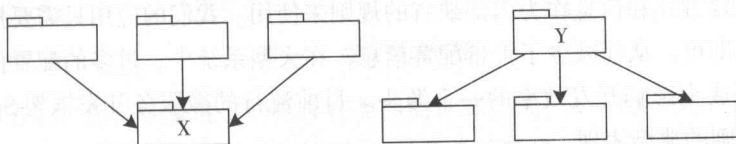


图4-1 稳定的X组件和不稳定的Y组件

3. 其他原则

除了面向对象和组件设计原则,业界还存在一批具有代表性的设计原则,包括但不限于:

(1) 合成/聚合复用原则

合成/聚合复用原则(Composite/Aggregate Reuse Principle, CARP)强调在一个新的对象里面使用一些已有的对象,使之成为新对象的一部分,而新的对象通过向这些对象的委派达到复用这些对象的目的。也即推荐首先使用合成/聚合,其次才考虑继承。合成/聚合使系统灵活,达到复用的目的。而使用继承时,要严格遵循里氏替换原则。有效地使用继承会有助于对问题的理解,降低复杂度,而滥用继承会增加系统构建、维护时的难度及系统的复杂度。所以合成/聚合复用原则在很多时候就体现为一句话,即组合优于继承。

(2) 迪米特法则

迪米特法则(Law of Demeter, LoD)认为一个软件实体应当尽可能少的与其他实体发生相互作用。这样,当一个模块修改时,就会尽量少的影响其他的模块,扩展也会相对容易。这是对软件实体之间通信的限制,它要求限制软件实体之间通信的宽度和深度。如果两个类不必彼此直接通信,那么这两个类就不应当发生直接的相互作用。如果其中的一个类需要调用另一个类的某一个方法的话,可以通过第三者转发这个调用。设计模式中的门面模式和调停者模式实际上就是迪米特法则的具体应用。

(3) 命令-查询分离原则

当一个方法返回一个值来响应一个请求,它就具有查询(Query)的性质;当一个方法要改变对象的状态,它就具有命令(Command)的性质。通常,一个方法可能是纯的Command模式或者是纯的Query模式,或者是两者的混合体。在设计接口时,如果可能,应该尽量使接口单一化,保证方法的行为严格的是命令

或者是查询，这样查询方法不会改变对象的状态，没有副作用，而会改变对象状态的方法不可能有返回值。这就是命令 - 查询分离（Command-Query Separation, CQS）原则。查询功能和命令功能的分离，有助于系统性能，也有利于系统的安全性。

（4）惯例优于配置原则

惯例优于配置（Convention over Configuration, CoC）原则简单讲就是将一些公认的配置方式和信息作为内部缺省的规则来使用。我们的应用只需要指定惯例外的信息即可，从而减少了大量配置信息，在大型系统中，过多的配置信息很多时候已经成为影响开发效率的一个源头。目前流行的微服务开发框架 SpringBoot 就是该原则的典型表现。

（5）关注点分离原则

关注点分离（Separation of Concerns, SoC）原则就是指在软件开发中，通过各种手段将问题的各个关注点分开。实现关注点分离的方法主要有两种，一种是标准化，另一种是抽象与包装。标准化就是制定一套标准，让使用者都遵守它，这样使用标准的人就不用担心别人会有很多种不同的实现，使自己的程序不能和别人的兼容。另一方面，不断地把程序的某些部分抽象并包装起来，也是实现关注点分离的好方法。诸如组件、分层、面向服务等概念都是在不同的层次上做抽象和包装以使得使用者不用关心它的内部实现细节。

4. 软件设计原则应用

本节以组件设计原则在 Dubbo 框架中的应用来展示如何在一个分布式服务框架中发挥指导性的设计作用。Dubbo（<http://dubbo.io/>）是 Alibaba 开源的一个分布式服务框架，在互联网行业应用和扩展十分广泛。Dubbo 的核心功能为我们进行分布式系统设计提供了两大方案，即高性能、透明化的 RPC 实现方案和服务治理方案。

（1）组件设计原则背后的量化标准

在组件设计原则中我们提到一个稳定抽象原则，即组件的抽象程度应该与其稳定程度保持一致。组件的稳定度可以用以下公式来衡量： $I = Ce / (Ca + Ce)$ 。其中 Ca 代表 Afferent Coupling，即向心耦合，表示依赖该组件的外部组件数量，而 Ce 代表 Efferent Coupling，即离心耦合，表示被该组件依赖的外部组件的数量。 I 代表 Instability，即不稳定性，它的值处于 $[0, 1]$ 之间。在前面的图 4-1 中，组件 X 的 $Ce=0$ ，所以不稳定性 $I=0$ ，说明它非常稳定。相反，组件 Y 的 $Ce=3$ ， $Ca=0$ ，所以它的不稳定性 $I=1$ ，说明它非常不稳定。图 4-2 展示的是一种更常见的场景，沿着依赖的方向，组件的不稳定性应该逐渐降低，稳定性应该逐渐升高。如果已经处于稳定状态的组件就不应该去依赖处于不稳定状态的组件。

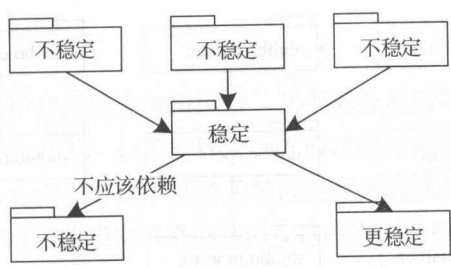


图4-2 稳定性的传递

另一方面，组件抽象度也同样存在类似的计算公式： $A = N_a / N_c$ 。其中 A 代表 Abstractness，即抽象度。Na 表示组件中抽象类的数量，而 Nc 表示组件中所有类的总和，这样通过对比 Na 和 Nc 就能得出该组件的抽象度。

正如图 4-2 所示，一个系统中多数的组件位于依赖链的中间，也就是说它们即具备一定的稳定性也表现出一定的抽象度。而在图 4-3a 中，如果一个组件的稳定度和抽象度都是 1，意味着该组件里面全是抽象类且没有任何组件依赖它，那么这个组件就没有任何用处。相反，如果一个组件稳定度和抽象度都是 0，那么意味着这个组件不断在变化，不具备维护性，这也是我们不想设计的组件。所以，在稳定度和抽象度之间我们应该保持一种平衡，在图 4-3a 中间的那个线就是平衡线。

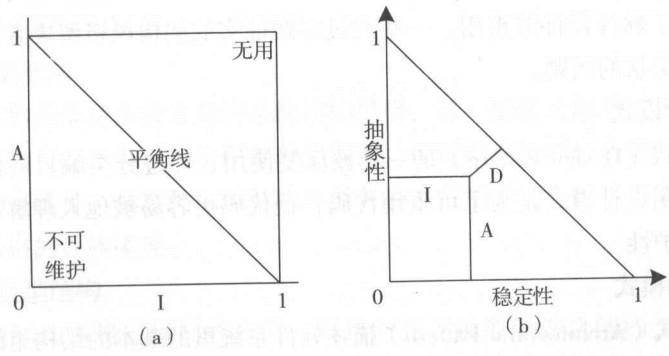


图4-3 稳定度和抽象度的平衡以及距离示意图

我们用距离（Distance）的概念来量化这种平衡，距离的计算公式： $D = \text{abs}(1 - I - A) * \sin(45)$ 。距离的图形化表示参考图 4-3b。

(2) Dubbo 中的分包原则

Dubbo 在设计过程中同样采用的是稳定抽象和稳定依赖等原则，图 4-4 展示的就是 Dubbo 中各个包之间的依赖关系，除了 dubbo.common 通用工具包之外，处于依赖关系底层的 dubbo.remoting 包和 dubbo.rpc 包是整个框架中的高层抽象。

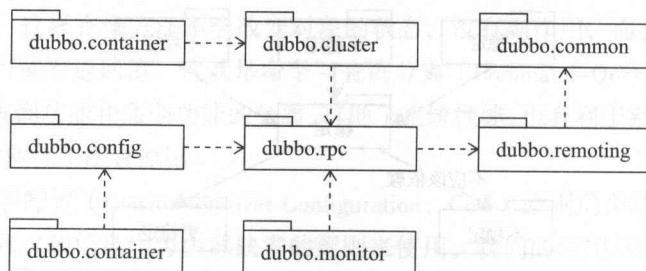


图4-4 Dubbo包依赖关系

通过 JDepend 等工具对 Dubbo 中的这些包依赖关系进行分析，我们可以得出 Dubbo 中的多数包结构在稳定性和抽象性之间达到了一种平衡^[8]。

4.1.2 技术理论的表现形式

前文介绍的软件设计原则是大而全的理论体系，为了更好地介绍这些技术理论，我们有必要从不同的角度发掘它们的表现形式，本书中对技术理论表现形式的理解体现在架构风格、设计模式、架构模式和架构模型等四个主要方面。

· 架构风格

架构风格（Architecture Style）描述某一特定应用领域中系统组织和表现的惯用方式。架构风格是软件技术理论的重要组成部分，对软件体系架构风格的研究和实践促进了软件设计的重用，一些经过实践证实的架构风格解决方案也可以可靠地用于解决新的问题。

· 设计模式

设计模式（Design Pattern）是一套被反复使用、经过分类编目的代码设计经验总结。使用设计模式是为了可重用代码，让代码更容易被他人理解，保证代码可靠性和维护性。

· 架构模式

架构模式（Architectural Pattern）描述软件系统里的基本的结构组织或纲要。架构模式提供一些预先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。一个架构模式常常可以分解成很多个设计模式的联合使用。

· 架构模型

架构模型则从另一个角度出发讨论软件系统如何满足功能性需求和非功能性需求，并展示架构设计过程中的视图、视角和所使用的模型，用来处理软件高层次结构的设计和实施。

以上四个维度构成了软件开发理论体系的主体内容，有些概念比较抽象化，本章后续内容将从各个维度出发具体讨论它们各自的特点以及典型应用场景。

4.2 架构风格

在技术理论体系中存在很多架构风格，其中部分通用架构风格可以按照其系统组织和表现形式进行分类，分类的效果一方面对软件系统的设计和实现起到良好的指导作用，另一方面也有助于理解并掌握主流开源系统的设计理念。本节将介绍软件开发过程中常见的架构风格类别以及这些类别中的代表性架构风格。

4.2.1 系统结构风格

软件开发都会经历从混沌到结构的过程。一方面，人、业务、复杂度与时间演进一般成正比，随着系统存在时间的推移，各个因素相互作用会导致系统复杂度呈指数化上升，理解、维护和改进系统比开发系统需要花费更大的代价，以至推倒重来、重复造轮子的事情在软件行业并不少见。出现这种现象的很大一个原因就在于对系统结构把握的不合理。可以认为粗粒度抽象和分离是设计过程中提供系统较高层次结构化的基本思路。

1. 系统结构划分的切入点

任何软件系统的设计都需要考虑系统结构，针对如何划分系统结构这个问题，有几点通用的思路，如系统如何与环境交互、系统处理流程如何组织、系统需要支持什么样的变化、系统的生命周期等。围绕这些问题，我们可以抽象出一系列系统结构相关的架构风格。

2. 分层结构

分层结构是最基本最常见的系统结构风格，每一层次之间通过接口与实现的契约方式进行交互，可以严格限制跨层调用，也可以支持部分功能的跨层交互以提供分层的灵活性。典型的三层结构以及各种在三层结构上衍生出来的多层结构就是这种风格的具体体现。

3. 交互型结构

交互型结构风格应用同样广泛，其目的在于抽象组件之间的交互关系，常见的有 MVC（Model View Controller）、MVP（Model View Presenter）、MVVM（Model View ViewModel）等表现形式。MVC 风格是目前 Web 开发领域的主流分层风格，图 4-5 即为 MVC 模式的基本结构，我们可以看到该图中 View 和 Model 之间存在直接交互，通过 Controller 我们也可以把这层直接交互关系转变成间接交互关系。

另外一种比较流行的 MVP 模式则更加明确的规定 Model 和 View 之间不应该存在直接交互，Presenter 中作为一种协调器同时保存着 View 和 Model 的引用，确保 Model 和 View 之间的数据传递通过 Presenter 集中进行。

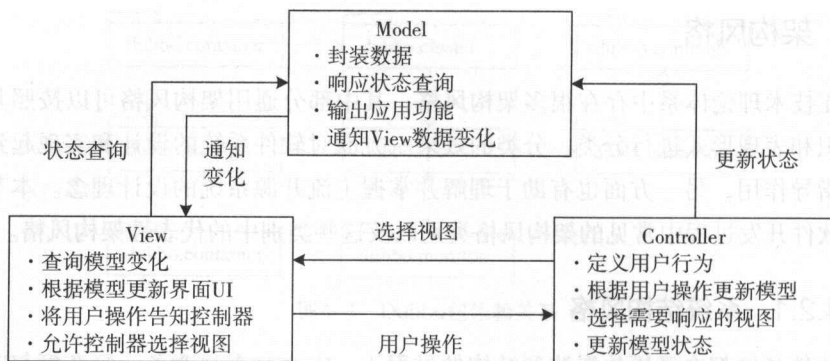


图4-5 MVC结构

4. 系统结构风格应用

系统结构风格应用广泛，以目前最流行的 Spring 框架而言就包含 Spring MVC Web 组件，该组件即是 MVC 风格的典型应用。Spring MVC 的整体架构如图 4-6 所示。

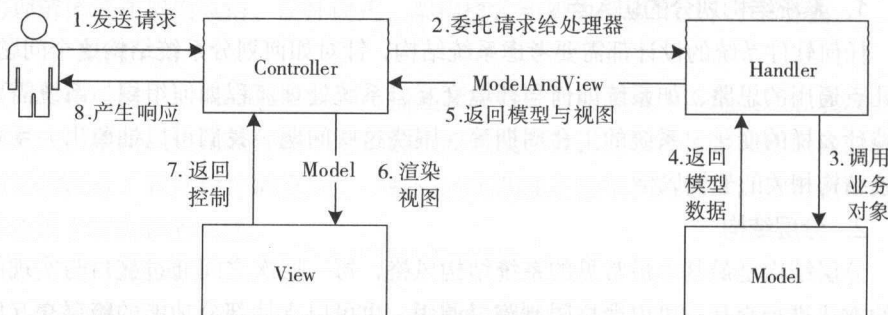


图4-6 Spring MVC架构

在 Spring MVC 中，首先用户发送请求到前端控制器（Controller），前端控制器根据请求信息（如 URL）来决定选择哪一个处理器（Handler）进行处理并把请求委托给它；处理器接收到请求后，进行功能处理，首先需要收集和绑定请求参数到一个对象，这个对象在 Spring Web MVC 中叫命令对象，并进行验证，然后将命令对象委托给业务对象进行处理；处理器处理完毕后返回一个 ModelAndView（模型数据和逻辑视图名）；前端控制器收回控制权，然后根据返回的逻辑视图名，选择相应的视图，并把模型数据传入以便视图渲染；前端控制器再次收回控制权，将响应返回给用户，完成整个流程。

4.2.2 数据流风格

前面介绍的系统结构风格主要面向组件之间存在相互调用关系的系统。而本

节介绍的数据流风格则从另一种系统构建需求切入，有些系统内部虽然同样存在各种组件，但这些组件之间各自独立，并不存在相互调用的关系。系统的行为由数据流控制，这种系统普遍存在于数据处理领域。

1. 顺序批处理风格

随着离线大数据技术的兴起，顺序批处理（Sequential Batch）风格得到了广泛应用。在顺序批处理系统中，各个组件互相独立，并且这些组件按照先后顺序处理，仅当一个组件运行彻底结束之后，下一个组件才能开始执行。而且在各个组件之间传输的是成批（Batch）数据，而不是以数据流的方式运行，所以称之为顺序批处理。一个典型的离线批量处理系统如图 4-7 所示，这是一个 ETL（Extract-Transform-Load，抽取-转换-装载）系统，我们可以看到每一个组件之间传输的是批量数据，这些批量数据通常以文本文件为存储媒介。

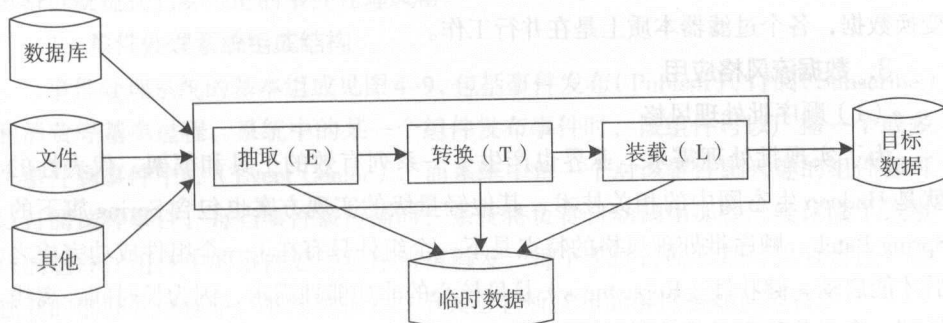


图4-7 离线批量处理ETL系统

顺序批处理风格中，每个组件都是互为独立的程序，只有上一步组件完成之后下一步组件才能开始。同时，数据作为一个整体批量传输。因为每个组件只能顺序进行，其性能取决于其包含的所有组件的处理时间总和。以上特点决定了其不适合应用于实时数据处理，而往往面向海量数据的离线处理场景。

2. 管道-过滤器风格

管道-过滤器（Pipe-Filter）风格代表另一种面向数据流的处理方式。顾名思义，管道-过滤器结构主要包括过滤器和管道两种元素（见图 4-8）。其中功能组件被称为过滤器（Filter），负责对数据进行加工处理。每个过滤器都有一组输入端口和输出端口，从输入端口接收数据，经过内部加工处理之后，传送到输出端口上。数据通过相邻过滤器之间的连接件进行传输，管道（Pipe）可以看作输入数据流和输出数据流之间的通路。

管道-过滤器结构中，不同过滤器之间不需要进行交互，独立的过滤器能够减小组件之间的耦合程度，可以很容易地将新过滤器添加到现有的系统之中，原

有过滤器也可以很方便地被改进的过滤器所替换以扩展系统的业务处理能力。

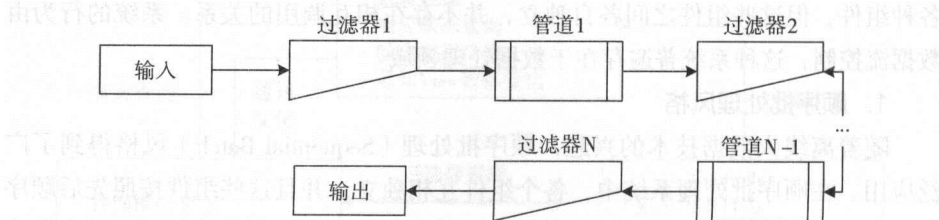


图4-8 管道-过滤器风格

虽然在管道-过滤器风格中，每个过滤器之间不存在相互调用，即每个组件的处理过程也保持独立，这点与顺序批处理风格一致。但两者在数据处理方式上有本质不同，在顺序批处理风格中，数据以批量方式传输，且只有上一个组件完成之后才能传输到下一组件。但在管道-过滤器风格中，过滤器以流对流的方式变换数据，各个过滤器本质上是在并行工作。

3. 数据流风格应用

(1) 顺序批处理风格

为了实现批处理需求，业界也产生了一系列有效的工具和框架，代表性的就是 Hadoop 生态圈中的相关技术，其他轻量级的实现方案也包含 Spring 旗下的 Spring Batch。顺序批处理风格的特点是下一个组件只有在上一个组件成功完成之后才能启动，健壮性（Robustness）是最核心的非功能性需求，因为长时间、离线处理、自动化等需求促使批处理的过程应该保持在非人工干预下能够有一定的智能化机制。

要想实现智能化，首先需要确保对任务的执行过程可跟踪并在产生失败时进行任务重启。健壮性的实现体现在以下三种策略：Skip 即忽略，对于类如文件中某个数字格式错误等非致命异常，我们认为直接忽略这些异常比停止 Job 更加合适；Retry 即重试，对于网络失败或数据库锁等瞬态异常，重试在很大程度上能够确保 Job 的正常执行；Restart 即重启，对于因为数据格式错误太多、业务处理异常等任务执行失败场景，Skip 会导致业务逻辑错误而 Retry 显然也不能解决问题，这时候就需要对问题进行修正后重新执行。以上三种健壮性策略可以组合使用，也可以根据特定业务规则在运行过程中把某种策略动态替换成其他策略，比如，当数据格式错误小于某个界限时，使用 Skip 是合适的，但超过这个界限就认为需要进行 Restart。

(2) 管道-过滤器风格

管道-过滤器风格的典型应用就是 Linux 命令执行过程。例如在 `cat input.txt | grep "hello" | sort > output.txt` 这句命令中，grep 命令在文件 input.txt 中搜索包含

“hello”的所有行，并且将它们打印出来，然后 sort 命令将上一步的结果内容进行排序，并通过 sort > output.txt 命令将排序后的结果输出到 output.txt 文件中。这些过程的每一步都相当于是通过过滤器对前一步的结果进行处理，然后通过管道进行串联。在 Linux 中，广泛使用管道 - 过滤器风格处理类似命令。

4.2.3 事件处理风格

在一些软件系统中会存在这样的需求，当系统中的某个组件因为用户操作或内部行为发布一个事件到事件中心，该组件知道这个事件在将来的某一个时间点会被其他某个组件所消费，但是并不知道这个组件具体是谁、也不关心什么时候被消费。同样，调用该事件的组件也不一定需要知道该事件是由哪个组件所发布。满足以上场景的系统代表着一种松耦合的架构，通常被称为事件系统，这种设计风格也就是我们要讨论的事件处理风格。

1. 事件处理系统组成结构

事件处理系统的基本组成见图 4-9，包括事件发布 (Publish)、订阅 (Subscribe) 和消费等基本过程。系统中的某一个组件发布事件时，该组件可以广播一个或多个事件到事件中心 (Event Center)，而系统中每一个对该事件感兴趣的组件都可以订阅这种事件。每当事件被传播时，系统将负责自动调用那些已经订阅了该事件的组件，组件中的事件处理程序将被触发。每个事件订阅者都可以有自己一套独立的事件处理程序，事件发布者并不关心它所发布的事件被如何消费。

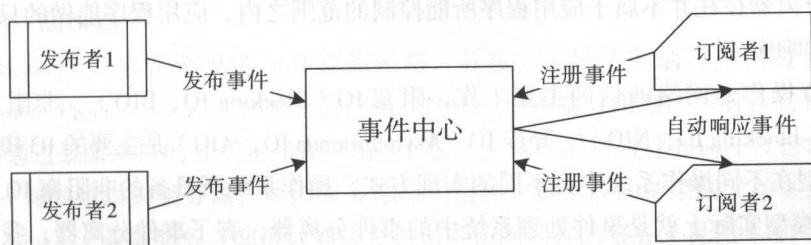


图4-9 事件处理系统组成结构图

事件作为一种传输媒介有两个主要特点，首先事件具备异步性和并发性，事件到达的时机是系统无法提前确定的；同时，事件一般都不止一种类型，一个系统中往往需要同时处理多种事件类型。因此，对事件的处理，我们同样需要设计一种抽象的事件处理策略。

2. 事件处理策略

基于图 4-9，事件处理根据是否包含独立的事件分派器 (Dispatcher) 可以分为两种主要的处理策略。

（1）无独立事件分派器的事件处理策略

无独立事件分派器的事件处理策略比较简单，即系统中任何一个组件都可以表示对某个事件感兴趣并订阅该事件。当事件发生时，这些事件仅会发布给那些已经订阅该事件的组件。从这一点上看，无独立事件分派器的事件处理与设计模式中的观察者（Observer）模式比较类似，适用于相对较小的应用系统。

（2）有独立事件分派器的事件处理策略

有独立事件分派器的事件处理策略则相对复杂，在事件发布者和订阅者之间加了一层事件分派器。在系统中添加事件分派器的主要目的在于增加控制力度，因为在大型事件系统中，存在多种不同类型的事件，订阅者对于事件的订阅可能存在一些业务规则而不是简单的通用订阅。另一方面，发布者和订阅者的数量较大的情况下，也需要由事件分派器进行统一管理，事件分派器作为中间层组件就能起到协调作用。

3. 事件处理系统应用

本节将讨论事件处理风格在网络通信这个常见主题中的应用。我们在前面已经看到事件处理系统的抽象可以参考分层思想，即把事件驱动系统按照事件的来源和处理过程分成三个层次：事件源（Event Source）、事件分派器和事件响应程序（Event Handler）。对于网络通信而言，套接口（Socket）即是事件产生的源头，操作系统级别的 select/poll/epoll 程序对应事件分离器，而业务系统中各种应用程序就是事件响应程序。通过这种抽象，我们认识到在网络通信模型中，事件源和事件分离器往往并不属于应用程序所能控制的范围之内，应用程序能做的只是对事件的响应。

IO 操作是网络通信的主要工作，阻塞 IO（Blocking IO，BIO）、非阻塞 IO（Non-blocking IO，NIO）、异步 IO（Asynchronous IO，AIO）是主要的 IO 模型。IO 模型在不同操作系统中有不同的实现方式，操作系统所具备的非阻塞 IO 和异步 IO 模型实际上就是事件处理系统中的事件分离器，有了事件分离器，我们就可以在此基础上实现针对业务的事件响应程序，也引出了针对事件处理的 Reactor 模式。

Reactor 模式^[9]定义事件循环（Event Loop），利用操作系统事件分离器支持单线程在一系列事件源上同步等待事件，这里有三个词需要注意，即单线程、一系列事件源和同步。Reactor 模式体现的是 IO 复用思想，支持多个事件源响应，而响应的方式并不是采用多个线程，而只是使用一个单线程构建事件循环，这个事件循环是一个死循环，一直阻塞等待事件的发生。当事件发生时，事件循环从操作系统提供的事件分离器中获取事件，并将事件逐个分发给对应的事件响应程序，后者对它的事件做出同步处理，这里的事件响应程序位于应用系统中，而且

事件处理同样也是一个同步的过程（见图 4-10）。Reactor 模式应用广泛，是实现诸如 Netty、Mina 等 NIO 通信框架的典型模式。

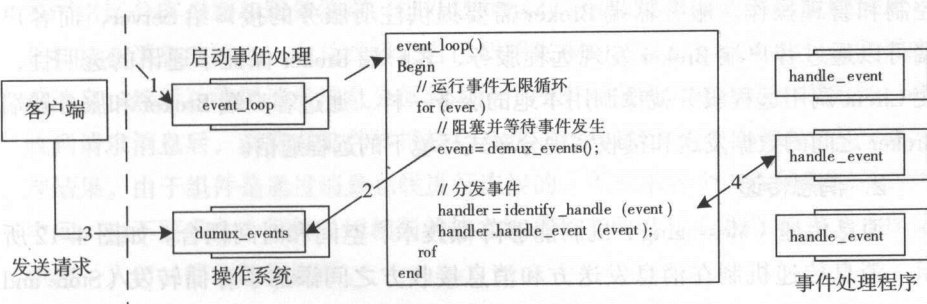


图4-10 Reactor模式

4.2.4 分布式风格

现代软件开发一大现状和问题就是容易形成信息孤岛，即每个业务部门和产品线根据自身的业务需求设计相应的架构体系并进行实现，这些独立的系统保存着业务系统的核心数据并运行于独立的进程环境中。涉及组织架构、团队文化、技术体系等因素，仅仅通过系统内部重构很大程度上只能缓解而不能消除信息孤岛的产生，这就需要使用某种手段进行业务服务整合，分布式架构风格的目的就在于此。分布式作为一种基础架构风格为不同系统之间的交互提供通信范式，从而有效屏蔽底层平台细节。分布式架构风格有以下三种主要的表现形式：

1. Broker

Broker^[9]是非常常见的一种架构风格，分布式系统之间通过远程过程调用（Remote Procedure Call，RPC）进行相互作用。服务端应用组件注册自己到 Broker，通过暴露接口的方式允许客户端接入服务。客户端则通过 Broker 发送请求，Broker 转发请求到服务端，调用服务端应用组件并将生成的结果回发给客户端。通过 Broker，业务服务之间可以通过发送请求访问远程的服务（见图 4-11）。

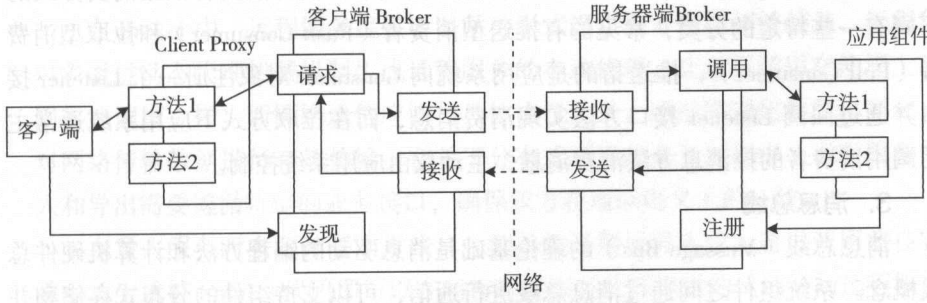


图4-11 Broker风格

Broker 中存在几个核心组件：Client，Server，Proxy 以及分别面向 Client 和 Server 的 Broker，其中最重要的 Broker 可以被看成请求转发器，同时也负责一些控制和管理操作。服务器端 Broker 需要提供注册服务的接口给 Server，而客户端可以通过客户端 Broker 发现远程服务，客户端 Broker 保证了通讯的透明性，使 Client 调用远程服务就像调用本地的服务一样。通过客户端 Broker 和服务器端 Broker 之间的数据发送和接收实现分布式环境下的远程通信。

2. 消息传递

消息传递（Messaging）机制能够降低技术、空间和时间耦合。如图 4-12 所示，消息传递机制在消息发送方和消息接收方之间添加了存储转发（Store and Forward）功能。存储转发是计算机网络领域使用最为广泛的技术之一，基本思想就是将数据先缓存起来，再根据其目的地址将该数据发送出去。显然，有了存储转发机制之后，消息发送方和消息接收方之间并不需要相互认识，也不需要同时在线，更加不需要采用同样的实现技术。紧耦合的单阶段方法调用就转变成松耦合的两阶段过程，技术、空间和时间上的约束通过中间层得到显著缓解，这个中间层就是消息传递系统（Messaging System）。

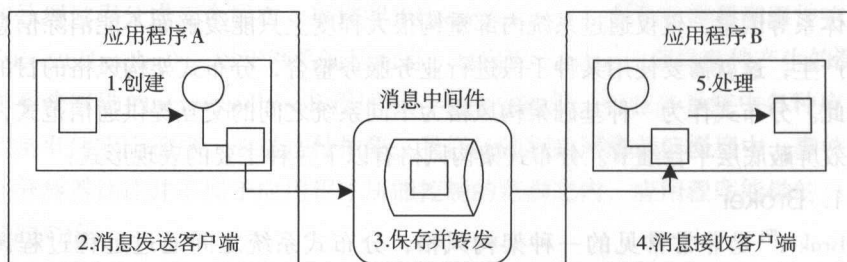


图4-12 消息传递机制

在消息传递系统中，消息的发送者称为生产者（Producer），负责产生消息，一般由业务系统充当生产者；消息的接收者称为消费者（Consumer），负责消费消息，一般是后台系统负责异步消费。生产者行为模式单一，而消费者根据消费方式的不同有一些特定的分类，常见的有推送型消费者（Push Consumer）和拉取型消费者（Pull Consumer），推送指的是应用系统向 Consumer 对象注册一个 Listener 接口并通过回调 Listener 接口方法实现消费消息，而在拉取方式下应用系统通常主动调用消费者的拉消息方法消费消息，主动权由应用系统控制。

3. 消息总线

消息总线（Message Bus）的理论基础是消息驱动的编程方法和计算机硬件总线概念。系统组件之间通过消息总线进行通信，可以支持组件的分布式存储和并发运行。消息总线是系统的连接件，负责消息的分派、传递和过滤，并返回处理

结果。

消息总线风格可以视为分布式消息传递风格的一种扩展和延伸。系统组件并不严格区分客户端和服务端，组件之间也不是通过消息通道进行直接交互，而是挂接在消息总线上，向总线登记自己所感兴趣的消息类型。生产者组件发出请求消息，然后总线把请求消息分派到系统中所有对此感兴趣的消费者组件，在接收到请求消息后，消费者组件将根据自身状态对其进行响应，并通过总线返回处理结果。由于组件是通过消息总线进行连接的，不要求各个组件具有相同的地址空间，也不要求各个组件采用相同的技术体系和实现机制（见图 4-13），而是提供统一入口，简化数据拦截。

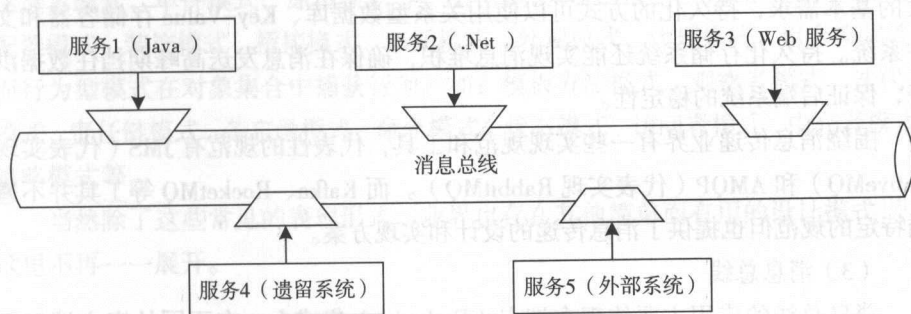


图4-13 消息总线风格

在组件之间，消息是唯一的通信方式。根据需要，消息总线对消息具备丰富的预处理功能，包括消息路由（Routing）、消息转换（Transformation）和消息过滤（Filtering）。这些预处理功能提供了高度扩展性，但同时也为系统的设计和实现带来了不可避免的复杂度。

4. 分布式风格应用

Broker、消息传递和消息总线这三种分布式风格都在分布式系统架构设计领域有广泛的应用，目前很多主流的分布式框架的背后就是以这三种风格为基础。

（1）Broker

在 Broker 中，远程服务提供者以某种形式提供服务调用相关信息，远程代理对象通过动态代理拦截机制生成远程服务的本地代理，让远程调用在使用上就如同本地调用一样。而网络通信应该与具体协议无关，通过序列化和反序列化方式对网络传输数据进行有效传输。在基于分布式环境的交互过程中，远程服务的导入和导出需要遵循特定的业务接口，确保双方在通信语义上的一致。

在导入服务时有两种基本方式，一种是编译期代码生成，通过调用前在客户端本地生成桩（Stub）代码即可以在运行时使用桩代码提供的代理访问远程服务，Web Service 中通过 wsdl 生成客户端代码就是这种方式的典型表现；另一种更常见

的方式是运行时通过动态代理 / 字节码的方式动态生成代码。

针对以上分布式交互的需求，业界提供了一大批优秀的基于 Broker 思想构建的分布式服务框架，如 Alibaba 的 Dubbo、Facebook 的 Thrift、Google 的 gRPC 等。

（2）消息传递

消息传递有两种基本模型，即发布 - 订阅（Pub-Sub）模型和点对点（Point to Point）模型。发布 - 订阅支持生产者消费者之间的一对多关系，是典型的推送消费者实现机制；而点对点模型有且仅由一个消费者，通过拉取或基于间隔性拉取的轮询（Polling）方式进行消息消费。消息持久化是消息传递系统实现存储转发的基本需求，持久化的方式可以使用关系型数据库、Key-Value 存储容器和文件系统。持久化存储系统还能实现消息堆积，确保在消息发送高峰期挡住数据洪峰，保证后端系统的稳定性。

围绕消息传递业界有一些实现规范和工具，代表性的规范有 JMS（代表实现 ActiveMQ）和 AMQP（代表实现 RabbitMQ）。而 Kafka、RocketMQ 等工具并不遵循特定的规范但也提供了消息传递的设计和实现方案。

（3）消息总线

消息总线的应用主要体现在端点（Endpoint）集成上。在不同的客户端和服务端系统中需要提取专门与消息通道发生直接交互的组件以便于业务模块解耦，这种组件一般就被称为端点。各种不同的技术体系所实现的客户端应用需要通过中间的消息总线进行交互，那么端点可以用来屏蔽这种技术差异性。

围绕消息总线和端点集成，业界具有代表性的消息总线工具有 Mule ESB、Apache Camel 和 Spring Integration。以 Spring Integration 为例，作为 Spring 家族中一员以及轻量级、松耦合集成框架，与现有 Spring 系统完美融合，支持和扩展主流系统集成模式，并提供众多基础性系统交互端点技术。Spring Integration 所提供的常见集成端点包括 File、FTP、TCP、HTTP、JDBC、JMS、JPA、Mail、MongoDB、Redis、RMI、Web Services 等。

4.3 设计模式

本书不是一本介绍设计模式的书，但我们认为对设计模式的了解有利于提升设计系统的能力，也有利于掌握一些目前主流开源框架的设计原理和理念。设计模式是软件开发技术理论的重要组成部分，本节先对常见的设计模式进行简要介绍，然后重点给出设计模式在开源框架中的应用场景。

4.3.1 设计模式

软件设计开发中使用设计模式的主要目标是通过再次使用已经获得的设计经验提高系统的灵活性。大部分系统中都会或多或少包含一些设计模式，很多时候实现同一个功能会有很多不同的方式，但使用设计模式往往能够提供更为优雅的实现方式。

依据设计模式的行为方式，我们可以将设计模式分为三种类型：创建型模式、结构型模式、行为型模式^[10]。

创建型模式以灵活的方式创建对象的集合，如：工厂模式、抽象工厂模式、建造者模式、单件模式、原型模式等；结构型模式代表相关对象的集合，如：适配器模式、装饰模式、桥接模式、享元模式、外观模式、代理模式、组合模式等；而行为型模式在对象集合中捕获行为，如：模板方法模式、观察者模式、迭代子模式、责任链模式、备忘录模式、命令模式、状态模式、访问者模式、中介者模式、策略模式等。

当然除了这些常见的表现形式，业界也存在其他简单而有用的设计模式^[11]，这里不再一一展开。

4.3.2 设计模式应用

掌握设计模式的难度在于每个模式看上去都能够理解但就是不知道如何应用，固然在日常开发过程中多尝试应用设计模式有助于提高自身的设计能力，但通过阅读源代码的方式来加深对设计模式的理解和掌握是一条捷径。目前非常主流的开源 ORM 框架 Mybatis (<http://www.mybatis.org/>) 中大量用到了设计模式，这里举几个典型的场景分析如何将设计模式应用到 Mybatis 框架设计中，包括建造者模式、工厂模式、模板方法模式、单例模式、装饰器模式等。

在 Mybatis 中存在一批以 Builder 结尾的类，这些类基本都是建造者模式的具体体现。例如 `SqlSessionFactoryBuilder` 其目的就是构建 `SqlSessionFactory`，在 Mybatis 环境的初始化过程中，会调用 `XMLConfigBuilder` 读取所有的 `MybatisMap-Config.xml` 和所有的 `*Mapper.xml` 文件，构建 Mybatis 运行的核心对象 `Configuration` 对象，然后将该 `Configuration` 对象作为参数构建一个 `SqlSessionFactory` 对象。而 `XMLConfigBuilder` 在构建 `Configuration` 对象时，也会调用 `XMLMapperBuilder` 用于读取 `*Mapper` 文件，然后 `XMLMapperBuilder` 会使用 `XMLStatementBuilder` 来读取和构建所有的 SQL 语句。在这些过程中，有一个相似的特点，就是这些 Builder 会读取文件或者配置，然后做大量的 XpathParser 解析、配置或语法的解析、反射生成对象、存入结果缓存等步骤，这些工作都不适合简单的通过构造函数来提供

初始化服务，因此大量采用了建造者模式来解决。

又如常见的工厂模式在 Mybatis 也体现在一系列以 Factory 结尾的工厂类，最典型的的就是 DefaultSqlSessionFactory 类，该 Factory 的 openSession 方法重载了很多个，分别支持 autoCommit、Executor、Transaction 等参数的输入来构建核心的 SqlSession 对象。

在 Mybatis 中，SqlSession 的 SQL 执行都是委托给 Executor 实现的，Executor 包含如图 4-14 中的类层结构，其中的 BaseExecutor 就采用了模板方法模式，它实现了大部分的 SQL 执行逻辑，然后把具体几个方法交给子类定制化完成。这几个子类的具体实现使用了不同的策略，体现了模板方法模式基于继承的代码复用技术。

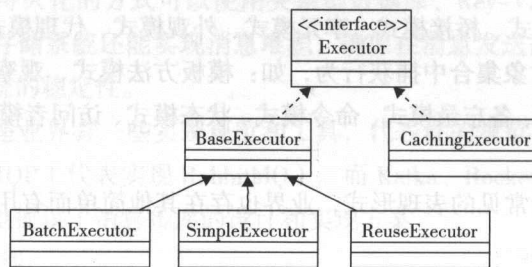


图4-14 Mybatis中Executor类层结构

针对单例模式，在 Mybatis 中有两个地方用到该模式，即 ErrorContext 和 LogFactory。其中 ErrorContext 是用在每个线程范围内的单例，用于记录该线程的执行环境错误信息，而 LogFactory 则是提供给整个 Mybatis 使用的日志工厂，用于获得针对项目配置好的日志对象。以 ErrorContext 为例，构造函数是 private 修饰，具有一个 static 的局部 instance 变量和一个获取 instance 变量的方法，在获取实例的方法中，先判断是否为空，如果是的话就先创建，然后返回构造好的对象。

装饰器模式在 Mybatis 主要应用在对缓存的处理上。在 Mybatis 中，缓存的功能由根接口 Cache 定义，数据存储和缓存的基本功能由 PerpetualCache 永久缓存实现，然后通过一系列的装饰器来对 Perpetual Cache 永久缓存进行缓存策略等方面的控制。用于装饰 PerpetualCache 的标准装饰器包括 FifoCache、LruCache、LoggingCache 等，通过类似 Cache cache = new LoggingCache(new PerpetualCache (“cacheid”)) 的方式实现装饰。

4.4 架构模式

在《面向模式的软件架构》第一卷^[12]中提到，根据问题的规模或抽象层次可以将软件设计模式分成三个层次，即架构模式、设计模式和习惯用法（Idiom）。

其中架构模式是一种高层模式，用于描述系统级的结构组成、相互关系及相关约束。对架构模式的选择是最基本的设计决策，将决定系统的基本架构以及后续的设计及开发活动。而设计模式是中层模式，是针对系统局部设计问题给出的解决方案。设计模式的选择对系统的基本架构没有影响，但在实现架构模式时，则可能采用多种设计模式。如在 4.2.1 中提到的 MVC 可以视为一种架构模式，该模式给出了一种交互式系统的架构设计，主要思想是实现业务逻辑和用户界面之间的分离。而为了实现这一设计理念，可以采用的主要设计模式包括观察者模式、组合模式和策略模式等多种设计模式。架构模式和设计模式被认为是与具体编程语言和工具无关的，而习惯用法则被认为是一种与具体实现方式相关的低层模式。习惯用法给出的解决方案通常与具体编程语言的某种语法机制相关。

另一方面，讲到架构模式，我们也需要理解它与架构风格之间的区别。架构模式基于子系统或模块及其之间的关系层次，描述了架构粗粒度的解决方案，着重描述系统的内部组织。架构风格描述的是系统组织方式的惯用方式，是系统组织性的设计，可以认为风格是模式的外在表现。架构模式的切入点在于找到系统中某一个结构需求并提供内部组织的抽象。虽然架构模式和架构风格在概念上有所不同，但在有些场景下往往并不容易也没有必要严格区分所使用的设计结构究竟是架构风格还是架构模式，正如前面提到的 MVC 风格，很多时候也可以被认为是一种架构模式。同时，较之架构风格，架构模式数量众多，本书也不会对所有的架构模式面面俱到，而是侧重介绍对于一个软件系统而言通常都会涉及的架构模式，并通过具体应用场景分析来给出这些架构模式更深层次的理解。

业界存在的架构模式非常之多^[12]，本节只挑选最具代表性的几个架构模式进行分析并介绍它们在不同场景下的应用。

4.4.1 微内核模式

1. 微内核模式简介

微内核（Microkernel）架构模式结构如图 4-15 所示，有时也被称为插件架构模式（Plug-in Architecture Pattern），通过插件向核心应用添加额外的功能，可以实现功能的独立和分离。

微内核架构包含两部分组件，即内核系统（Core system）和插件（Plug-in Component）。微内核架构的内核系统通常提供系统运行所需的最小功能集，插件是独立的组件，用来向内核系统增强或扩展额外的业务能力。

那么插件是什么？插件一般由以下几部分组成：插件暴露的接口（一般称为 API），插件内部实现，插件扩展点以及插件配置。其中插件扩展点我们一般设计为 SPI（Service Provider Interface，服务提供接口）。

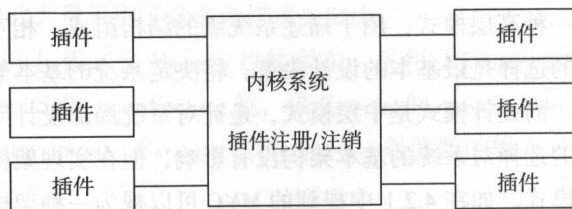


图4-15 微内核架构模式

微内核模式的本质是管理插件以及协调插件之间的调用。插件本身是一个很大粒度的扩展点，可以整个被替换。这样整个系统就是由一个微内核加很多插件组成一个具备很强扩展性的系统。使用微内核设计，对系统进行升级，显然只要用新插件替换旧插件，不需要改变整个系统架构。

2. 微内核模式在 Dubbo 中的应用

微内核模式在 Dubbo 中应用广泛，通信框架 Mina、Netty 和 Grizzly，序列化方式 Hession、JSON，传输协议 Dubbo、RMI 等都是这一架构模式的体现。我们可以通过简单的配置就能对这些具体实现进行排列组合构成丰富的运行时环境。微内核模式提供的是一种解决扩展性问题的思路，Dubbo 中实现这一思路的正是 SPI 机制。

JDK 提供了服务实现查找的一个工具类 `java.util.ServiceLoader` 来实现 SPI 机制。当服务的提供者提供了服务接口的一种实现之后，在 jar 包的 `META-INF/services/` 目录同时创建一个以服务接口命名的文件，该文件里配置着实现该服务接口的具体实现类。而当外部程序装配这个模块的时候，就能通过该 jar 包 `META-INF/services/` 里的配置文件找到具体的实现类名，并装载实例化，完成模块的注入。基于这样一个约定就能很好地找到服务接口的实现类，而不需要在代码里硬编码指定。

Dubbo 提供专门的 `@SPI` 注解，只有添加 `@SPI` 注解的接口类才会去查找扩展点实现，查找位置包括 `META-INF/dubbo/` 和 `META-INF/services/`，而 `META-INF/dubbo/internal/` 中则定义了各项用于供 Dubbo 本身使用的内部扩展。举例来说，前面提到 Dubbo 对传输协议提供了 Hessian、Dubbo 等多种实现，Dubbo 内部通过扩展点的配置确定使用何种机制。在 `dubbo-rpc-default` 工程和 `dubbo-rpc-hessian` 工程中，我们在 `META-INF/dubbo/internal/` 目录下都发现了 `com.alibaba.dubbo.rpc.Protocol` 配置文件，但里面的内容分别指向了 `com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol` 和 `com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol` 类，意味着当我们引用某个具体工程时，通过该工程中的配置项就可以找到相应的扩展点实现。

对于 Dubbo 的整体架构，微内核作为一种架构模式只负责组装功能，而所有功能都通过扩展点实现。Dubbo 自身功能也是基于这一机制实现，所有功能点都可被用户自定义扩展和替换。所有扩展点定义都通过传递携带配置信息的方式在运行时传入 Dubbo，确保整个框架采用一致性数据模型。

4.4.2 资源管理

所谓资源（Resource），在软件架构设计过程中有很多表现形式，如数据库会话、网络连接、分布式服务和组件等都可以认为是系统的资源，都需要进行管理，而性能、可伸缩性、灵活性代表着资源管理的基本需求。资源管理包含一组模式，代表着资源从创建到销毁的整个过程，广泛应用于面向系统构建过程中的各个场景。

1. 资源生命周期管理模式

资源生命周期（Lifecycle）管理模式通常表现为一个容器（Container），该容器提供外部系统访问资源的入口。容器中保存着各种资源对象，客户端可以通过查询获取特定资源并使用该资源，资源对象本身的创建、删除等生命周期由容器自动完成，对客户端完全透明。

2. 资源有效性管理模式

资源有效性管理模式的目的在于最大化复用现有的资源以满足特定场景下的资源利用需求。系统中一般都会存在数据库会话、网络连接等非托管资源（Unmanaged Resource），这些资源的生命周期不受容器管理，而且往往需要较大的创建和销毁成本。如果对这些资源不加控制，任由应用程序无限制的使用，很容易导致出现性能和可用性问题。针对这些非托管资源，资源有效性管理模式的基本思路是“池化”操作，即初始化一个具有一定容量的资源池（Resource Pool），处于资源池中的对象可以重复利用，从而避免应用程序每次使用资源都需要创建和销毁新对象。“池化”操作的具体实现方式和应用场景有很多，这里介绍常见的对象池和线程池。

（1）对象池

对象池（Object Pool）模式管理一个可替代对象的集合，组件从池中借出对象，用它来完成一些任务并当任务完成时归还该对象，被归还的对象接着满足请求，不管该请求是来自同一个组件还是其他组件。

对象池适用于类的实例化过程开销较大、类的实例化的频率较高的场景，同时类的实例可重用于交互且参与交互的时间周期有限。对象池节省了创建类的实例的开销，但存储空间随着对象的增多而增大，通常具有一个控制池大小的特定策略以达到时间和空间的平衡。

资源池 (2) 线程池

假设服务器使用一个线程完成一项任务，创建线程时间为 T_1 、在线程中执行任务的时间为 T_2 、销毁线程时间为 T_3 。如果 T_1+T_3 远大于 T_2 ，则可以采用线程池来提高服务器性能。

线程池 (Thread Pool) 主要由四个核心组件构成，包括线程池管理器 (Pool Manager)，用于创建并管理线程池，包括创建线程池、销毁线程池、添加新任务；工作线程 (Pool Worker)，线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；任务接口 (Task)，每个任务必须实现的接口，以供工作线程调度任务的执行，主要规定任务的入口、任务执行完后的收尾工作、任务的执行状态等；任务队列 (Task Queue)，用于存放没有处理的任务，提供一种缓冲机制。

线程池的实现机制可以参考 JDK 自带的线程池，在 JDK 中，Executor 接口表示线程池，execute(RunnableTask) 方法用来执行 Runnable 的任务。Executor 类包含生成各种类型线程池的静态方法，而 ExecutorService 声明如 shutdown() 等管理线程池的方法，为用户提供访问线程池的统一入口。

3. 资源创建 / 获取 / 释放策略模式

资源创建策略模式的核心思想是将一个复杂对象的构建算法与它的组件及组装方式分离，使得构建算法和组装方式可以独立应对变化。复用同样的构建算法可以创建不同的表示，不同的构建过程同样可以复用相同的组件组装方式。

设计模式中的部分创建型模式可以归为资源创建策略模式的具体表现，最典型的就是 Builder 模式^[10]。从效果上讲，资源创建策略模式隐藏对象的内部表示。同时，将构造代码和表示代码分开，每一个具体对象包含了创建和装配该特定对象的所有代码，提供不同的具体对象就可以构建出不同的 Builder 结果。

当我们获取资源时，根据获取时机、完整性的不同希望获取资源的内容也会有所不同，资源获取策略模式就是处理这方面的需求。延迟获取 (Lazy Acquisition)、尽快获取 (Eager Acquisition)、部分获取 (Partial Acquisition) 等都是常见的资源获取策略模式。以延迟获取为例，延迟获取又叫延迟加载 (Lazy Load)，该机制的提出是为了避免一些无谓的性能开销，即仅当在真正需要数据的时候，才真正执行数据加载操作。

资源释放策略同样影响着资源的生命周期和使用方式，除了面向对象中普遍使用的自动垃圾回收 (Auto Garbage Collection) 机制之外，引用计数 (Reference Count) 机制也被广泛应用于资源管理场景。

引用计数的概念其实比较简单，就是每个对象都有一个计数用来表明有多少其他的对象目前正保留着对它的引用 (Reference)。对象 A 想要引用对象 B，A 就把 B 的 Reference Count 加 1，而当 A 结束了对 B 的引用，A 就把 B 的 Reference

Count 减 1。当没有任何对象再引用 B 时, B 的 Reference Count 就减为 0, B 就被清除, 内存就被释放。清除 B 的时候, 被 B 所引用的对象的 Reference Count 也可能减小, 也可能使它们被清除。

4. 资源管理模式应用

(1) Spring 中的资源管理

Spring 框架实际上就是一个容器, 管理着所有注册在 ApplicationContext 中 JavaBean 的生命周期。Bean 是 Spring 管理的基本单位, 在 Spring 中所有的组件都是 Bean, 任何的 Java 对象, 诸如各种 DataSource、SessionFactory 等 Java 组件都可被当成 Bean 处理, 即使这些组件并不是标准的 JavaBean。

在基于 Spring 的应用程序中, 整个应用中各层的对象都处于 Spring 的管理下, Spring 负责创建 Bean 实例, 并管理其生命周期。Bean 在 Spring 容器中运行时, 无须感受 Spring 容器的存在, 一样可以接受 Spring 的依赖注入, 包括 bean 属性的注入, 合作者的注入及依赖关系的注入等。

(2) ORM 框架中的资源管理

以 Hibernate、Mybatis 为代表的各种 ORM 框架中都提供了延迟加载功能。以 Hibernate 为例, 实体的集合属性默认会被延迟加载, 实体所关联的实体默认也会被延迟加载。Hibernate 通过这种延迟加载来降低系统的内存开销, 从而保证 Hibernate 的运行性能。

在 Hibernate 中是通过代理 (Proxy) 机制来实现延迟加载。Hibernate 从数据库获取某一个对象数据时、获取某一个对象的集合属性值时, 或获取某一个对象所关联的另一个对象时, 由于没有使用该对象的数据 (除标识符外), Hibernate 并不从数据库加载真正的数据, 而只是为该对象创建一个代理对象来代表这个对象, 这个对象上的所有属性都为默认值; 只有在真正需要使用该对象的数据时才创建这个真正的对象从数据库中加载它的数据。由于 JDK 的动态代理只能代理接口, 而我们在应用 Hibernate 的时候需要代理类, 所以一般用 cglib 来实现动态代理类的功能。

(3) 数据库连接池

由于数据库连接是一种稀缺资源, 所有在数据库连接的管理上一般都会使用连接池 (Connection Pool)。数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接, 当需要建立数据库连接时, 只需从“缓冲池”中取出一个, 使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况, 为系统开发、测试及性能调整提供依据。Java 中开源的数据库连接池有很多, 常见的包括 C3P0、DBCP、proxool、Druid 等。

4.4.3 服务定位

1. 服务定位基本概念

系统集成 (System Integration) 是应用系统架构设计的一个重要课题, 无论何种行业和应用, 系统都可能需要集成对第三方服务的访问。这些第三方服务可能来自外部供应商, 也可能来自于同一组织的不同团队, 更广义上来讲, 同一团队内部也可能需要进行服务的发现和整合, 以便不同技术体系结构下的各个模块和组件之间的集成。

系统集成需要解决的主要问题是如何获取并管理第三方服务。第三方服务同样需要业务迭代和版本更新, 当这些服务的实现发生了变化, 那么涉及集成部分的代码可能需要重构, 如果有些用户层面的代码还不能被直接访问的话, 整个重构的成本就会很大。服务定位 (Service Locator) 模式想要解决的问题就是解耦服务提供者 (Service Provider) 和用户, 应用程序无须直接访问具体的服务提供者类, 而解决方案就是服务注册 (Registration) 和发现 (Discovery) 机制。

图 4-16 中包含了服务定位模式的几个核心组件:

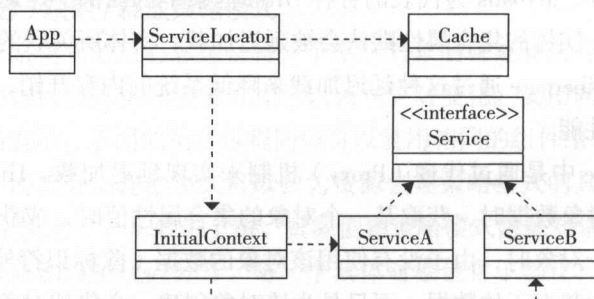


图4-16 Service Locator模式结构

(1) 服务 (Service), 实际处理请求的服务, 可以包含不同的实现。对这种服务的引用可以在类似 JNDI (Java Naming and Directory Interface, Java 命名和目录接口) 服务器的中央注册中心中查找。

(2) 上下文 (Context), 上下文中带有对要查找的服务的引用, 如 JNDI 使用 InitialContext 作为上下文容器。

(3) 服务定位器 (Service Locator), 服务定位器是通过类似 JNDI 服务器的中央注册中心查找并获取服务, 同时可以根据需要对服务进行缓存。

(4) 缓存 (Cache), 缓存存储服务的引用以便复用。

(5) 客户端 (App), App 是通过 Service Locator 调用服务的对象。

服务定位模式本质上体现的还是解耦思想, 支持服务动态升级, 提高了系统的可维护性。

2. 服务定位应用

在大型分布式系统中，由于涉及服务的发布者和消费者之间大量的调用关系，服务的定位显得尤为重要，因此主流的分布式服务框架都提供了类似注册中心的机制作为服务提供者和服务消费者进行交互的媒介，充当着服务注册和发现服务器的作用。

一个典型的注册中心模型参考图 4-17，图中的注册中心应该具备发布 - 订阅功能，体现在服务提供者可以根据服务的元数据发布服务，而服务消费者则通过对自己感兴趣的服务进行订阅并获取包括服务地址在内的各项元数据。发布 - 订阅的功能还体现在数据变更推送，即当注册中心服务定义发生变化时，主动推送变更到服务的消费者从而实现服务路由。由于服务提供者和服务消费者同时依赖于注册中心，就需要确保数据一致性，在任何时候服务提供者和消费者都应该看到同一份数据。同时服务消费者具备缓存功能，当注册中心不可用时，就可以同步本地缓存中的路由信息获取服务提供者的地址并实现远程调用。

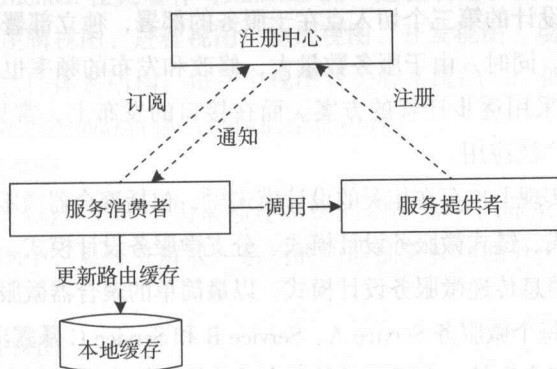


图4-17 注册中心模型图

再以 Dubbo 为例，其注册中心包含 Multicast 注册中心、Zookeeper 注册中心、Redis 注册中心和 Simple 注册中心等多种实现方式。无论使用何种实现方式，其基本的模型和工作流程与服务定位模式保持高度一致。

4.4.4 微服务架构

Martin Fowler 指出^[5]，微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小型服务，服务之间相互协调和配合，为用户提供最终价值；每个服务运行在其独立的进程中，服务与服务之间采用轻量级通信机制互相沟通；每个服务都围绕着业务进行构建，并且能够被独立部署到生产 / 类生产环境；尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应该根据业务上下文，

选择合适的语言、工具进行构建。我们可以对这段话进行总结和提炼，认为微服务具备业务独立、进程隔离、团队自主、技术无关轻量级通信和交付独立性等“微”特性。

微服务架构设计首要的切入点就是服务之间的集成方式，即需要保证集成 API 的技术无关性，不要选择对服务具体实现有技术性限制的集成技术。采用技术无关的集成接口，充分融合技术多样性意味着我们在特定场景下不应该使用类似 Dubbo 这种采用私有协议、重量级的通信框架，而应该尽可能使用类如 RESTful 的轻量级通信方式进行服务集成，确保服务易于使用，消费方可以使用多种技术实现集成。

微服务架构设计的第二个切入点在于服务建模，尽可能明确领域的边界。我们可以充分利用下一节中将要介绍的领域驱动设计方法，通过识别领域 / 子域中的模块和服务、判断这些模块和服务是否独立、考虑提升某些模块和服务的层次并建立充血领域模型。

微服务架构设计的第三个切入点在于服务的部署，独立部署单个服务而不需要修改其他服务。同时，由于服务数量大，修改和发布的频率也可能很高，接口变化管理上通常采用逐步迁移的方案。而在接口的发布上，常见的 API Gateway 等模式也会得到广泛应用。

在微服务的实现上也存在相关的设计模式^[14]，包括聚合器微服务设计模式、代理微服务设计模式、链式微服务设计模式、分支微服务设计模式、数据共享微服务设计模式、异步消息传递微服务设计模式。以最简单的聚合器微服务设计模式为例（见图 4-18），每个微服务 Service A、Service B 和 Service C 暴露出轻量级的 REST 接口，然后通过聚合为某一个页面或某一个功能提供数据处理和展示服务。

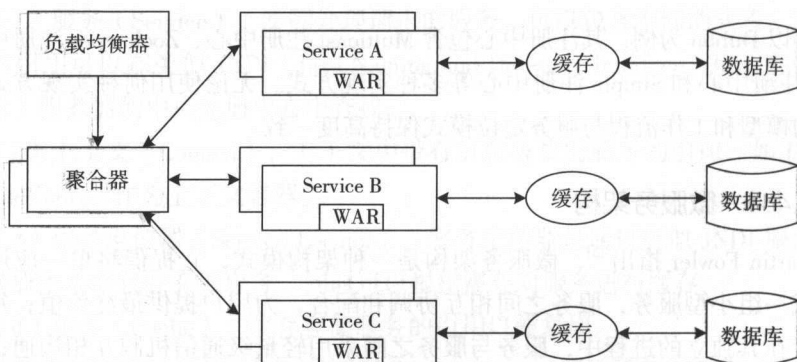


图4-18 聚合器微服务设计模式

4.5 架构模型

所谓模型 (Model), 是通过主观意识借助实体或者虚拟表现, 构成客观阐述形态和结构的一种表达目的的事物。在软件开发领域, 模型一词和架构一样被广泛应用于各种场合, 从模型本身的概念上讲, 架构模型本质是对架构本身的一种描述方式, 或者说代表一种视图 (View)。架构设计是一个抽象的过程, 借助模型的力量, 我们就可以表述抽象内容。

4.5.1 架构视图

架构视图是对从某一视角或某一切入点上看到的系统所做的简化描述, 描述中涵盖系统的某一特定方面, 而省略了与此方面无关的实体。系统架构视图代表性的有 4+1 架构视图和六维架构视图。

1. 4+1 架构视图

Philippe Kruchten 在其著作《Rational 统一过程引论》提出了一个“4+1”视图模型^[15], 从逻辑视图、进程视图、物理视图、开发视图、场景视图等 5 个不同的视图来描述软件体系结构。每一个视图只关心系统的一个侧面, 5 个视图结合在一起才能反映系统的软件体系结构的全部内容。

(1) 逻辑视图

逻辑视图 (Logic View) 用来描述系统的功能需求, 即系统所应该提供的功能。在逻辑视图中, 系统分解成一系列的功能抽象和功能分析, 这些主要来自问题领域。

(2) 进程视图

进程视图 (Process View) 考虑一些非功能性的需求, 如性能和可用性。它解决并发性、分布性、系统完整性、容错性的问题, 以及逻辑视图中的主要功能如何与进程结构相配合在一起执行, 即定义逻辑视图中的各个类的具体操作是在哪一个线程中被执行。所以进程视图侧重系统的运行特性并服务于系统集成人员, 方便后续开展性能测试。

(3) 物理视图

物理视图 (Physical View) 主要描述硬件配置。服务于系统工程人员, 解决系统的拓扑结构、系统安装、通信等问题。主要考虑如何把软件映射到硬件上, 也要考虑系统性能、规模、可靠性等。物理架构主要关注系统非功能性的需求, 如可用性、可靠性、容错性、吞吐量和可伸缩性。

(4) 开发视图

开发视图 (Development View) 描述了在开发环境中软件的静态组织结构, 即

关注软件开发环境下实际模块的组织，服务于软件编程人员。将软件打包成一系列组件或子系统，这些组件和子系统可以组织成分层结构，每个层为上一层提供良好定义的接口。

(5) 场景视图

场景视图 (Scenarios View) 综合所有的视图，用于刻画组件或子系统之间的相互关系，将四个视图有机地联系起来。该视图可以描述一个特定的视图内的组件或子系统关系，也可以将这些关系延伸到不同视图间的组件或子系统。

四种视图的元素通过一组重要场景进行无缝协同工作，在某种意义上场景是最重要的需求抽象。4+1 架构中各个视图的关系见图 4-19，可以看到场景视图是其他视图的冗余，这也是名称中 +1 的由来。

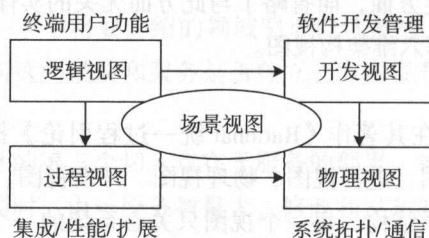


图4-19 4+1视图的关系

如果我们用 UML 来表示 4+1 视图，则场景视图往往对应用例图，逻辑视图对应类图和对象图，开发视图对应类图和组件图，进程视图对应顺序图、协作图、状态图、活动图和组件图，部署视图对应部署图。

2. 六维架构视图

六维架构视图^[6]为我们提供了六大视图，图 4-20 展示了这六大视图以及架构设计与这些视图之间的关系，所有视图都是围绕架构设计展开，但又各自具备侧重点。通过完备的架构视图，系统架构就从一种抽象的概念转变成能够供干系人触碰的软件实体。

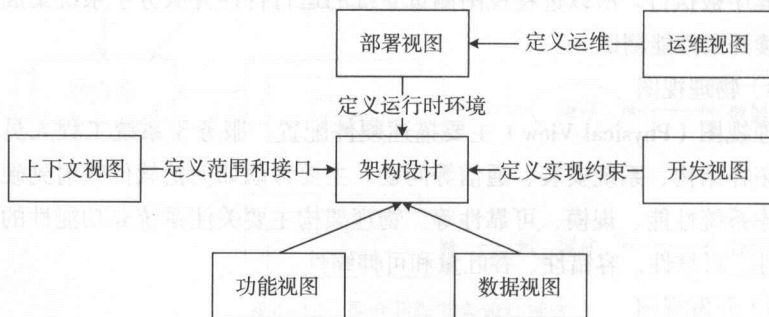


图4-20 架构设计与视图

我们基于定义、切入点和架构设计这一建模方式对每个视图进行展开。

（1）上下文视图

所谓上下文（Context）指的就是一种环境，上下文视图描述系统与环境之间的关系、依赖和交互，包含了各种当前环境中数据及其操作。通常，上下文包含在特定的场景中，所以有时候我们也可以把场景（Scenario）这个词视同系统的上下文。

架构设计方面，上下文视图总结我们所设计的架构背后究竟是怎么样的一个系统，包括系统本身、外部实体和相关接口。例如对于一个基于电商系统业务的上下文视图而言，其内部包含账户系统、支付系统、物流系统等核心功能子系统，同时也需要和各种第三方系统进行集成，相关数据都存储在本地数据库中，用户通过电商系统门户可以获取各个内部和外部子系统所提供的服务，从而提供了一幅完整的系统功能范围、外部系统集成和用户交互的上下文视图。

（2）功能视图

功能视图描述系统运行时功能元素及其职责、接口和交互关系，从定义上看，功能视图和上下文视图有一定的重合之处，但功能视图脱离环境，描述的是系统组件定义以及各个组件之间的交互关系而不是业务场景分析，所以对于功能视图而言，我们结合组件设计思想进行理解，可以把功能视图看作是组件之间的接口和依赖关系。

功能视图的切入点就是从功能出发，包括系统的内部结构和外部接口，推导出该系统所需的各个组件及其依赖关系。内部结构取决于系统建模和架构分析的结果，而外部接口受系统集成模式和实现技术的约束。

（3）数据视图

业务系统软件通过数据来承载结果，大多数实现过程都是围绕数据展开。数据视图描述系统存储、操作、管理和分发数据的方式，是系统中核心业务数据的一种载体和表现形式。

数据架构建模有几种典型方式，包括静态数据建模、数据流建模和数据状态建模。这些数据模型代表着数据在不同场景下的不同表现形式以及发挥的作用。在UML中，类图、流程图和状态图分别可以作为静态数据模型、数据流模型和数据模型的建模工具。

（4）开发视图

开发视图描述支持软件开发过程的架构。在所有视图中，该视图最接近于系统设计和具体实现方案，是架构设计中面向技术的核心视图。

系统设计和开发包含通用的体系结构和设计模式，其中系统模块的合理组织、软件复用技术的应用、使用设计和测试的标准化以及如何进行代码组织都是开发

视图常见的切入点。

上述切入点需要结合具体的业务需求并采用特定的架构设计模型方能展现成开发视图，模块结构模型是开发视图中比较容易实现且易于展示的一种模型，图4-21 就是一个涉及用户和商品管理的电商系统中所展示出来的模块结构图，采用了UML中的包图作为特定展示媒介。从图中我们可以看到在架构设计和系统开发中时常需要考虑的一些设计模式，如系统分层（领域层、业务逻辑层和表现层）以及这些层次之间的依赖关系。

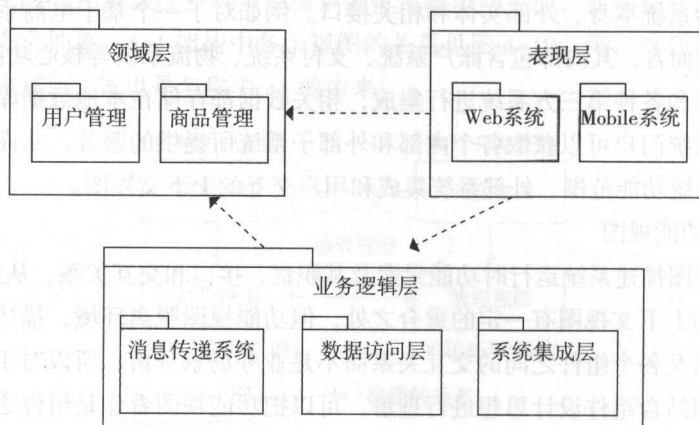


图4-21 开发视图示例

（5）部署视图

部署视图描述系统部署的环境，以及系统与其中元素的依赖关系。部署视图的切入点一般都比较程式化，系统部署所需的运行时平台、硬件设备和软件服务、第三方软件需求和网络需求是该视图的主要考虑点，通常这些考虑点已经包含了常见业务系统部署的各个方面。

当下服务化架构大行其道的背景下，系统服务之间的调用关系远比传统意义上单块模式复杂，部署视图的重要性得到显著提升。UML中，我们可以使用部署图对系统部署架构进行建模。

（6）运维视图

运维视图描述当系统运行在生产环境时如何进行运维、管理和支持，其目的是为了保证服务的稳定性和可用性，并在系统出现运行时故障时能够进行容错和快速恢复。运维视图并不属于系统架构设计的核心视图，该视图也通常由专业的运维人员进行开发和维护。

运维视图和部署视图一样，切入点通常都比较程式化，如建立隔离的生产环境、运行时的功能/数据迁移、状态/性能监控、集中式或分布式的配置管理、

数据和系统的备份和还原以及提供各项技术支持等都是常见的运维要求。

六大视图之间虽然各自表现架构的某个方面，但也存在依赖关系，图 4-22 描述的是视图之间的依赖关系。从图中可以看到，开发视图作为唯一没有被依赖的视图处于架构设计的低端，正如同编码开发处于软件系统工程的下游一样；上下文视图顾名思义处于中间位置，为多个视图提供上下文环境信息；功能和数据视图之间耦合度较高，分别构成了独立的视图系统，因此往往共同存在和发展，类似的还有部署和运维视图。

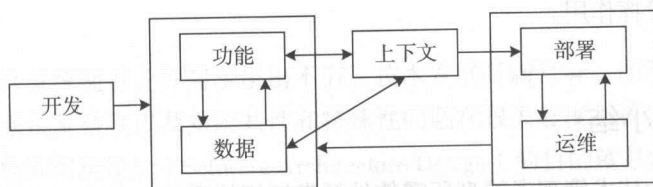


图4-22 六维视图之间的依赖关系

4+1 视图和六维架构视图实际上是对同一事物的不同理解和表述方式，其本质是一致的，我们从上面的描述中也不难看到它们两者之间的共性。

4.5.2 领域模型

领域(Domain)是对现实世界问题的一种统称，即是一个组织的业务开展方式，体现业务价值，所以领域模型(Domain Model)的目的是进行业务建模。通过分析业务场景和构建信息模型的方式展现系统对业务的理解和抽象，实现从现实世界的问题到抽象的问题的转变。

领域驱动设计^[17]为我们提供了一种软件开发方法，强调开发人员与领域专家协作交付业务价值，强调把握业务的高层次方向，也强调系统建模工具和方法以满足技术需求。领域驱动设计思想的核心就是认为系统架构应该是业务架构和技术架构想结合的一种过程，并提供了一系列的设计相关工具和模式确保实现这一过程。

开发人员对领域的思考方法体现在设计的维度(Design Dimension)上。在领域驱动设计中，有两个主要的设计维度，即设计的策略维度和设计的技术维度。

1. 设计的策略维度

设计的策略维度关注如何设计领域模型以及对领域模型的划分，其目的在于清楚界定不同的系统与业务关注点。策略维度是一个面向业务、具备较高层次的设计维度，偏重于业务架构的梳理以及考虑如何把业务架构和技术架构相结合的问题。

2. 设计的技术维度

设计的技术维度关注技术实现，从技术的层面指导我们如何具体地实施领域驱动，关注基于技术设计工具按照领域模型开发软件。显然，技术维度偏向于技术实现，体现了技术架构的设计和展现方式。

设计的策略维度和技术维度相结合提供了一套通用的建模语言和术语，展示基于领域驱动的架构设计方法和实现领域驱动设计的各项关键技术。领域模型是业务架构和技术架构的整合体，在处理复杂业务逻辑、保持系统边界、防止架构腐化等方面发挥作用。

4.6 本章小结

本章作为技术管理者转型所需软件开发知识体系的一个重要组成部分，介绍了软件开发过程中的理论知识和相关示例。

软件开发首先需要掌握的是一系列软件设计原则，包括面向对象原则、组件设计原则以及其他常见的设计原则。随后，我们梳理了软件技术理论的多种表现形式，包括架构风格、设计模式、架构模式和架构模型。

5 架构设计

现代软件系统随着业务需求层出不穷、技术发展日新月异、团队规模快速扩张等因素，系统复杂度以及系统共性和特殊性问题在很大程度上决定了软件开发的成败。而软件架构设计（Software Architecture Design）的目的就是对系统进行高度抽象，通过一系列设计活动在最大程度上降低系统复杂度，解决系统中存在的各种共性和特殊性问题的。

架构设计是整个软件开发过程中的核心工作之一，那么软件架构设计究竟是怎样一种工作内容？本书对这一问题的回答通过图 5-1 来展示，我们认为架构设计是技术理论、应用场景和工具框架的一种组合体。

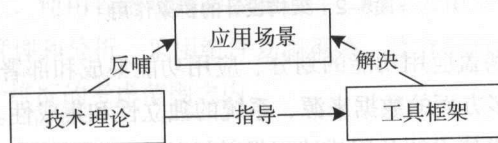


图5-1 架构设计工作的组合

我们已经在上一章中详细介绍了技术理论，包括软件设计的基本原则、架构风格、设计模式、架构模式和架构模型，并给出了一部分典型的示例。应用场景指的就是软件开发过程中所碰到的各种问题以及业务发展所提出的技术要求，架构设计的本质是满足需求，这里需求包括业务需求，也包括技术需求。工具框架是架构设计活动中技术人员最容易把握的切入点，通过应用业界主流的工具框架，就能找到解决问题的基本方法和思路。

显然，技术理论、应用场景和工具框架三者之间是相辅相成的，技术人员想要解决应用场景中的具体问题就会使用工具框架，而对工具框架的理解和把握首先需要深入理解其背后的技术理论，而技术理论不断发展有时候也会反哺应用场景。

围绕架构设计，本章在前一章技术理论的基础上，对应用场景和工具框架展开进一步探讨。

5.1 架构设计的层次和维度

架构设计是一套解决问题的方法论和工程实践，可以认为引入方法论和工程实践进行系统架构演进是一个原型→发现/改进→再发现/再改进的过程，这显然已经不是一个纯粹的技术问题，而是一项涉及多个软件开发层次和维度的系统工程。

5.1.1 架构设计的层次

架构设计被认为是从问题领域到解决方案的一种桥梁（见图 5-2），从图中我们可以看到架构设计活动与代表问题域的需求分析活动以及代表解决域的软件开发活动都有直接的交集，连接着两个软件开发的核心领域。从图 5-2 中我们可以看到架构设计的两个层次，即面向问题领域的业务架构（Business Architecture）和面向解决方案的技术架构（Technical Architecture）。

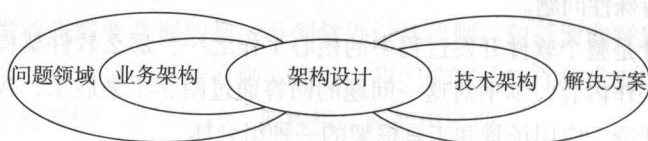


图5-2 架构设计的桥梁作用

业务架构设计涵盖应用功能的划分、应用功能集成和部署，主要关注点在于复杂的业务逻辑、多方面的数据来源、系统的独立性和集成性。

技术架构则提供技术的分层描述以及关键技术的方案，主要关注点在于系统运行时的各种设计要素，如高并发场景下的性能、可用性以及数据的安全性。

架构设计本质是满足业务需求，在现实中绝大多数场景下，业务架构驱动着技术架构，而不是反其道而行。

5.1.2 架构设计的维度

对于架构设计，我们从另一个视角审视就可以发现其最重要的几个设计维度，这些维度关注于架构设计最终的质量和结果。关于架构设计的维度或要素的划分，业界也有很多不同的说法，在本书中，我们认为以下四个维度对于架构设计而言至关重要：

1. 可扩展性

可扩展性（Extendibility）是指系统在经历不可避免的变更时足够灵活，针对提供这样的灵活性所要付出的成本进行平衡的能力。

对于可扩展性，架构设计上重点在于梳理系统的变化并把它们抽象成扩展点，

并通过对这些扩展点创建可扩展的接口，尽量使用基于业务标准的扩展点技术手段确保系统具有较高的可扩展性。

2. 性能

性能的要求体现在系统在其指定的性能状况下执行，以及将来需要时提供增长的处理能力。我们可以从核心功能响应时间、系统吞吐量、部署架构的可伸缩性、性能问题的可预测性和峰值负载等方向判断系统是否存在性能问题并找到相应的解决方案。

架构策略上，也有很多针对性能问题的设计方案，对核心业务链路和活动进行分解并把串行操作转变成并行化流程、对需要重复执行的处理过程进行优化、重用资源和结果、使用异步处理、转换数据强一致性为弱一致性等都可以在一定程度上提升系统的性能。

3. 可用性

可用性提供系统在需要时能够完整地提供服务，并有效处理影响系统可用性故障的能力。可用性的规划和实现需要先明确服务的类型，对不同类型的服务其可用性要求不尽相同。系统升级、系统备份、灾难恢复等也都需要有对应的实施计划。

架构设计策略上，使用容错硬件和容错软件、确保采用主流的集群和负载均衡机制、加强日志管理和分析、采用组件复制策略、建立完整的备份和灾难恢复解决方案都属于这个维度的考虑范围之内。

4. 安全性

安全性体现的是控制、监控和审计对资源的访问性和执行能力，以及从安全漏洞中恢复的能力。需要进行安全性控制的内容通常称之为资源（Resource），能访问资源的人或系统称为访问主体（Subject），控制安全性就是根据不同的访问主体对不同的资源进行精细化控制，包含建立完善的用户权限管理系统并提供相应安全策略。

对用户进行身份认证（Authentication）、授权（Authorization）访问、通过加密解密等确保信息保密性和完整性、提供类似单点登录（Single Sign On, SSO）的安全性管理平台、使用第三方安全性基础框架等都是安全性架构设计的常见手段。

综合考虑架构设计的层次和维度，我们不难发现，系统的可扩展性偏向于业务架构，而其余的性能、可用性和安全性则更多偏向于技术架构。但在具体设计过程中，有些思路 and 理念对业务架构和技术架构设计同样有效。在本章后续内容中，我们分别从业务架构和技术架构这两个主要层次出发，并结合各个设计维度展开讨论。

5.2 系统业务架构设计

业务架构设计就是通过一套方法论对产品或项目所涉及的业务需求进行业务边界划分、整合和扩展的过程。在这个过程中，我们需要考虑的切入点有以下三个主要子过程：

- 系统拆分

考虑如何将一个大而全的系统拆分成若干个子系统或子模块的过程，这些子系统和子模块需要有明确的业务边界。

- 系统集成

考虑如何将系统拆分所生成的一系列子系统和子模块进行整合的过程，这个过程同样需要明确集成的业务边界以及组织策略。

- 系统扩展

考虑在已具备系统拆分和系统集成的基础上，如何快速有效的添加新业务，确保系统的整体架构不需要做过多调整的过程。

5.2.1 系统拆分

任何软件系统的发展都是从简单到复杂、从集中到分散的过程。在系统构建初期，我们习惯于构建单一、内聚和全功能式的系统，因为这样的系统就能满足当前业务的需求。而当系统发展到一定阶段，集中化系统已经表现出诸多弊端，功能拆分和服务化思想和实践就会被引入。而当系统继续演进，团队规模也随之增大，由于分工模糊和业务复杂度的不断上升，系统架构逐渐被腐化，直到系统不能承受任何改变，也就到了需要重新拆分的阶段。推倒重来意味着重复从简单到复杂、从集中到分散的过程，这就是系统架构的轮回（见图 5-3）。

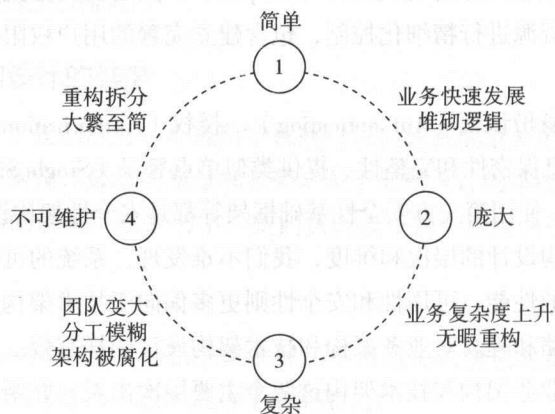


图5-3 架构的轮回

架构轮回给我们的启示就在于将所有东西放在一个系统中是不好的, 软件系统的关注点应该清晰划分, 并能通过功能拆分降低系统复杂性。在本节中我们关注的是业务拆分, 业务拆分要解决的问题就是如何找到拆分的合理边界, 为此领域驱动设计为我们提供了一种系统化的拆分方法。

1. 面向领域的系统拆分方法

所谓领域 (Domain), 即是对现实世界问题的一种统称, 是一个组织的业务开展方式, 体现一个组织所做的事情以及其中所包含的一切业务范围和所进行的活动。可以简单认为, 我们所要设计的每一个特定的业务系统就是一个领域。而为了清晰划分一个领域中的业务边界, 领域驱动设计中又包含子域 (Sub Domain)、聚合 (Aggregate) 和界限上下文 (Boundary Context)。

(1) 子域

针对如何进行系统划分, 领域驱动设计给出了子域的概念。子域作为系统拆分的切入点, 其来源往往取决于系统的特征和拆分的需求, 如核心功能、辅助性功能、第三方功能等。

子域的划分虽然因系统而异, 但通过对子域的抽象, 我们还是可以梳理出通用的分类方法。业界比较认可的分类方法认为, 系统中的各个子域可以分成核心域、支撑子域和通用子域三种类型, 其中系统中的核心业务属于核心域, 专注于业务的某一方面的子域称为支撑子域, 可以用于整个业务系统且作为一种基础设施的功能可以归到通用子域。当然, 我们可以根据需要在子域建立子域的抽象模型, 为了描述方便, 本节后续内容以上述的分类方法标记系统子域。

(2) 聚合

一个子域中包含若干聚合。聚合的核心思想在于将关联减至最少有助于简化对象之间的遍历。聚合的组成有两部分, 一部分被称为根 (Root) 实体, 是聚合中的某一个特定实体; 另一部分描述一个边界, 定义聚合内部都有什么。

聚合具备如下固定规则, 根实体具有全局标识、外部系统只能看到根实体、只有根实体才能直接通过数据进行查询获取, 其他对象必须通过聚合内部关联的遍历才能找到。这些固定规则都是为了减少复杂关系下对象遍历的次数, 明确系统边界。参考图 5-4 中的聚合示意图, 左半部分代表的是没有采用聚合概念的对象遍历图, 我们可以看到任何对象都能两两进行交互, 所以对象都处于同一个边界中; 而右半部分显然有所不同, 通过聚合思想把系统划分成三个边界, 每个边界里面包含一个聚合, 图中与外部边界直接关联的就是聚合中的根对象。我们可以看到只有根对象之间才能进行直接交互, 其他对象只能与该聚合中的根对象进行直接交互。以图中的 8 个对象为例, 通过聚合可以把最多 2^8-1 次对象直接交互减少的 2^3-1 次。

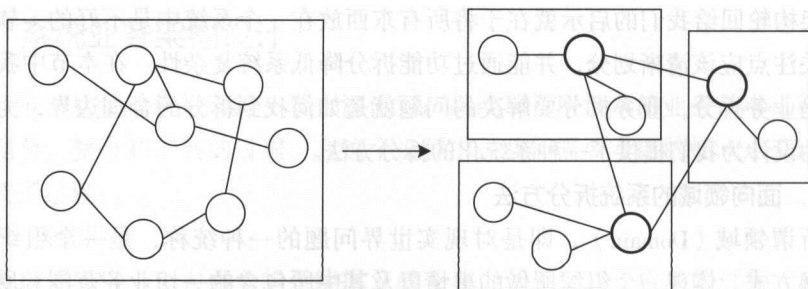


图5-4 聚合示意图

(3) 界限上下文

子域存在于界限上下文中，这里的界限指的是每个模型概念、属性和操作在特定边界之内具有特定的含义，这些含义只限于该界限之内。假如存在 A、B 两个上下文，其中 A 上下文和 B 上下文中都存在 User 对象，但是 B 上下文中的 User 对象可能不同于 A 上下文中的 User 对象，而 B 上下文中 Account 对象可能基于 A 上下文中的 Role 对象，这时候我们就会发现界限的划分能在很大程度上影响系统的设计和实现。

子域、聚合和界限上下文三者之间的关系如图 5-5 所示。该图根据业务功能的特性把整个系统拆分成四个主要的子域，分别包含一个核心子域，两个支撑性子域以及一个通用子域，每个子域都有其界限上下文，各个界限上下文之间可以根据需要有效整合从而构成完整的领域。

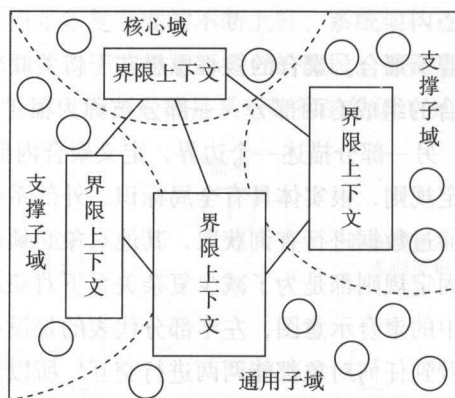


图5-5 子域、聚合和界限上下文三者之间的关系

2. 系统拆分策略

根据子域和界限上下文概念，我们就可以对系统进行拆分。系统拆分的策略可以因地制宜，但根据业务进行系统拆分是本书所推崇的方法。其他常见的拆分

策略也包括根据技术架构、根据开发任务分配以及一个团队负责一个上下文。

根据技术架构拆分系统违背了业务架构驱动技术架构的原则,在对业务梳理尚不完善、系统的策略设计尚不健全的情况下就考虑技术架构和实现方法,往往会导致返工,在不断的系统修改中腐化架构。

根据开发任务分配同样不是一个好主意,在系统拆分过程中实际还没有到具体开发资源和时间统筹的阶段,开发任务自然也无从谈起。

但是一个团队一个上下文策略有时候反而是一种有效的拆分策略。团队的构建方式可以是职能团队(Function Team)也可以是特征团队(Feature Team),前者关注于某一个特定职能,如常见的服务端、前端、数据库、UI等功能团队,而后者则代表一种跨职能(Cross Function)的团队构建方式,团队中包括服务端、前端等各种角色。上下文的构建以及界限的划分是一项跨职能的活动,如果团队组织架构具备跨职能特性,可以安排特定的团队负责特定的上下文并统一管理该上下文对应的界限。

5.2.2 系统集成

系统拆分之后,我们要考虑的问题就是如何对拆分后的功能进行组装,对于这个问题,基本的思路就是系统集成。站在高层次的架构分析角度,任何一个系统都可以处在其他系统的上游(Upstream),也可以位于其他系统的下游(Downstream)。所以系统集成实际上就是将上游系统与下游系统进行整合共同完成某一项业务的过程。

1. 系统集成的思路

从上下游关系这一角度讲,系统集成的范围比系统拆分的范围更为广泛,正如上节所述,在我们自身设计的系统中,我们需要通过子域、界限上下文等方式对系统进行拆分,那么反过来也自然需要在子域、界限上下文提供系统集成的方法。另一方面,很多业务场景下还需要与其他外部的系统进行集成才能完成特定的业务。所以系统集成具有同时面向内部和外部的双重性需求。

(1) 两种系统集成需求

面向内部的系统集成采用的方式就是面向领域的基础方式。如图5-6所示,假设通过领域拆分得到三个上下文A、B和C,A上下文同时位于B、C上下文的上游,B上下文相对A而言处于下游但相对C而言处于上游,C上下文则处在整个系统的最下游。

面向外部的系统集成则往往并不会十分复杂。通常,我们在设计上需要把某一个外部依赖所影响的范围限制在一个上下文中,图5-6中也展示了带有外部依赖的上下文集成关系,可以看到在B上下文中添加了对外部系统X的系统集成需求。

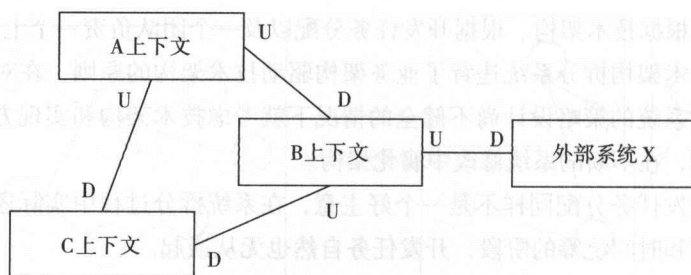


图5-6 上下文关系示例

（2）系统集成的组织关系

涉及系统之间的集成，无论是内部集成还是外部集成，现实中普遍存在三种最基本的组织关系。

第一种是供应商（Vendor）关系，即上游/下游关系的具体体现，客户方依赖供应商提供的服务才能构建自身系统。供应商关系是主流但不是最好的组织关系，因为处于下游的客户方系统受处于上游的供应商影响巨大。

第二种关系是合作关系，即分别处于上游和下游的两个上下文团队共同进退，双方通过制定合理的开发计划确保上下文集成工作能够顺利开展。合作关系是比较理想的关系，尽管并不一定能够有类似的条件。

第三种关系则是我们所不希望看到的，称之为遵奉者（Conformist）关系，即上游由于利益关系等因素并不想或没有能力推动系统集成，那下游只能妥协或另谋他路。

2. 系统集成的模式

系统集成模式的基本思路在于两点，一点是解耦，一点是统一。这两点对于内部集成与外部集成同样有效。

解耦比较容易理解，即两个系统集成时，一方面在于技术实现上的依赖性，支持异构系统的有效交互；一方面需要把关注于集成的实现与业务逻辑的实现相分离，确保集成机制的独立性。而统一的含义在于一致性，即上游系统应该定义协议，让所有下游系统通过协议访问，确保在数据传输接口和语义上各个上下文之间能够达成一致。

（1）系统集成的基本模式

针对以上两种思路，我们可以分别抽象出两种最基本的集成模式，防腐层（AntiCorruption Layer, ACL）和统一协议（Unified Protocol, UP）。防腐层强调下游系统根据领域模型创建单独一层，该层完成与上游系统之间的交互，从而隔离业务逻辑，实现解耦。统一协议则是提供一致的协议定义，促使其他系统通过协议访问。显然，防腐层模式面向下游系统而统一协议面向上游系统。

以内部系统集成为例，在对任何子域和上下文进行提取时，确保从组织关系和集成模式上对上下文集成进行抽象。图 5-7 即是图 5-6 上下文关系在集成方案上的一种表现形式。

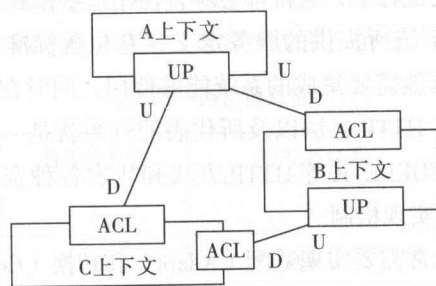


图5-7 上下文集成

（2）系统集成的扩展模式

有时候事情并没有看上去那么简单，特别是对于外部系统集成。上下游系统的数据结构需要统一、分布式环境下有效通信需要保证、数据需要根据业务需求进行过滤和路由等都是常见的基础要求。同时，对于复杂的业务场景，系统集成通常作为一个单独组件进行维护，应用业务组件如何与系统集成组件进行交互也是需要考虑的问题。

关于数据的过滤，我们可以通过设置合适的过滤器（Filter），而关于数据如何到达目标应用系统，我们可以使用路由器（Router）在一个位置上维护上下游目标地址并基于业务数据本身或上下文进行路由。转换器（Transformer）用于异构系统之间进行数据适配，数据结构、类型、表现形式、传输方式都是潜在的需要转换的对象。上述关于数据的过滤、路由、转换等功能构成了系统集成更深层次的需求，满足这些需求的系统就是服务总线（Service Bus）系统。4.2.4 节分布式风格中所讨论的消息总线就是服务总线系统的抽象。

3. 系统集成的技术

业界关于系统集成存在一些主流的模式和工程实践，包括文件传输（File Transfer）、共享数据库（Shared Database）、远程过程调用（RPC）和消息传递（Messaging）。这四种主流的集成模式各有优缺点。文件传输方式最大的挑战在于如何进行文件的更新和同步；如果使用数据库，在多方共享的条件下如何确保数据库模式统一是一个大问题；RPC 容易产生瓶颈节点；而消息传递在提供松耦合的同时也加大了系统的复杂性。关于这些基础模式的探讨可参考《企业集成模式》^[18]。

RPC 和消息传递面对的都是分布式环境下的远程调用，远程调用区别于内部方法调用，一方面网络不一定可靠和存在延迟问题，另一方面集成通常面对的是一些

异构系统。我们的思路是尽量采用标准化的数据结构并降低系统集成的耦合度。

（1）系统集成的基本实现技术

在系统集成的基本模式中，我们提到系统间集成的基本模式有两种，即防腐层和统一协议。在实现过程中，通常都会综合使用这些模式。

统一协议模式为系统所提供的服务定义一套包含标准化数据结构在内的协议，开发该协议以便其他需要集成的系统能够使用，同时在新有集成需求时对协议进行改进和扩展。HTTP 方法以及所代表的资源就是一种统一协议，我们可以通过 GET/PUT/POST/DELETE 等 HTTP 方法和代表各种资源的 Resource URI 构建 RESTful 风格的集成实现机制。

而防腐层的建立通常要实现适配（Adapt）和转换（Convert）。考虑这样一个场景，当上游系统通过基于 RESTful 风格的实现统一协议时，下游系统为了能够使用该服务，就需要考虑如何获取通过 HTTP 作为通信媒介的数据以及如何将这些数据转换为该系统自身所能识别的业务数据。图 5-8 展示了以上场景下防腐层的实现过程。

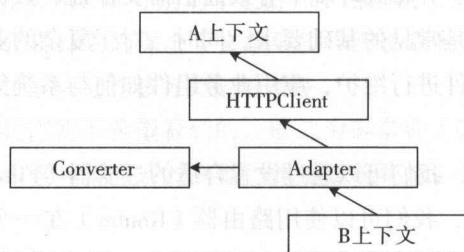


图5-8 防腐层示例

（2）企业服务总线

企业服务总线（Enterprise Service Bus, ESB）本质上是一种系统集成组件，用于解决分布式环境下的异步协作问题，可以看作是对消息传递系统的扩展和延伸。对系统集成扩展模式的剖析引出了服务总线的整体解决方案（见图 5-9），本节内容围绕这些问题对服务总线中的核心组件展开讨论。

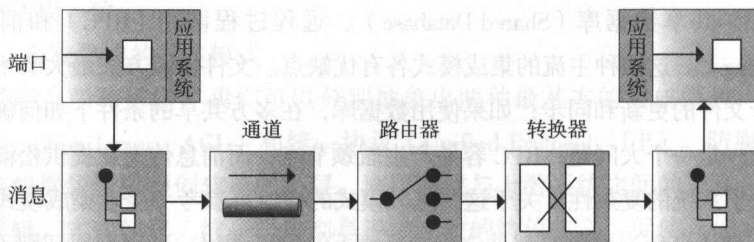


图5-9 服务总线解决方案

· 消息路由的实现

消息路由要考虑一次处理单条 / 多条消息、路由结果面向一个 / 多个目标、路由是否有状态等问题。围绕上述三个问题所得出的答案，我们加以排列组合可以得到多种路由器的表现形式（见表 5-1），其中有状态的路由器指的就是需要根据消息传递的上下文确定路由结果，通常涉及多个消息，具有较高的复杂性。

表5-1 路由器总览

路由器	消费消息数	发布消息数	有无状态
内容路由器	1	1	无
过滤器	1	0或1	无
接收表	1	0或n	无
分解器	1	n	有
聚合器	n	1	有
重排器	n	n	有

内容路由器（Content-based Router）是最简单的路由器，即通过消息的内容决定路由结果，这里的消息内容包括输入消息的消息头属性值、消息体类型以及各种对消息体内容的自定义的业务规则，通过内容路由器可以产生一对一的路由效果。

接收表（Recipient List Router）面向 1 对多的路由需求，当对同一消息进行路由时，特定场景下可能会产生多个路由结果。

过滤器（Filter）的目的是决定是否将消息流转到下一个环节，如果满足一定过滤条件，则该消息将不会产生任何路由结果，过滤条件同样可以包括复杂的业务流程性内容。

分解器（Splitter）的典型应用场景是消息包含多个元素，而每个元素处理方式不同，这时候我们可以把原始消息分解成多个消息，并通过复制关联标识符（CorrelationId）等公共属性的方式实现分解后消息的关联。

聚合器（Aggregator）往往和分解器一起使用，是分解的逆过程，将独立而又相关的消息组织成整体进行处理。聚合器的实现具有典型的状态性，因为独立消息的个数、到达顺序、相关性等因素都依赖于消息传递上下文。

分解器和聚合器组合与 Hadoop 中 Map/Reduce 算法有异曲同工之妙，对于数量较大的计算可以通过路由的策略并行执行，这也是解决类似问题的一种思路。

重排器（Resequencer）应该是最复杂的一种路由器实现，目的是为了将一批乱序到达的消息还原到按照顺序排列的效果。顺序是一种典型的状态性表现形式，

要想实现顺序就必须确保消息之间可比较。

· 消息转换的实现

消息转换器（Transformer）解决服务总线中数据如何在异构系统之间进行适配的问题。服务总线常用于异构系统之间的交互和集成，通过转换可以消除异构系统之间由于数据格式所导致的依赖。最基本的转换思路就是通过一种自定义机制进行两种数据结构之间的映射，但有些场景下我们也需要实现基本数据结构转换，并对输入数据结构进行内容的扩充和过滤。

内容扩充器（Enricher）就是往消息中扩充新的数据，扩充的数据来源可以来自计算、外部环境和第三方其他系统，扩展的对象可以是消息的消息头也可以是消息体，所以内容扩充器一般可以分成消息头扩充器（Header Enricher）和消息体扩充器（Payload Enricher）。内容过滤器是内容扩充器的对称操作，内容过滤的目的是去除消息中的某一部分而不是在消息传递过程中过滤掉该消息。

· 消息端点的实现

当应用程序与消息系统进行交互时，从系统设计的角度讲我们希望应用程序中的业务代码和用于消息传递的非业务代码耦合度尽量低，也就是说应用程序应该封装对消息系统的访问接口，消息网关（Gateway）就是用来实现这方面的需求。消息网关中应该只包含业务领域层面的接口定义而不应该出现任何和消息传递技术相关的内容。

（3）企业服务总线示例

企业服务总线在软件系统集成中应用广泛，以移动医疗系统为例就可以借助于企业服务总线完成与医院系统之间的整合。图 5-10 展示了移动医疗系统预约挂号模块中获取挂号信息功能的业务场景，用户通过手机 APP 连接到移动医疗系统中的挂号服务，而挂号服务则根据用户的挂号选择导向目标医院。针对具体某家医院，挂号服务需要通过系统集成的手段获取该医院的挂号详细信息并返回给用户。这个过程中，由于涉及与外部各个医院之间的信息传递和交互，我们就可以通过企业服务总线中的各个组件对其建模并实现。

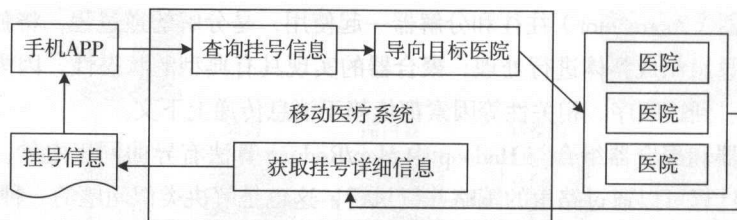


图5-10 移动医疗系统集成示意图

图 5-11 是基于企业服务总线的移动医疗系统集成实现效果图，我们综合应用了网关、通道、内容扩充器、接收表等组件完成系统集成建模。网关用于剥离移动医疗系统后台服务与集成服务之间的耦合，通道用于传递用户请求，而使用内容扩充器可以将用户的挂号请求目标医院信息放到消息头部分而不用修改消息体内容，内容路由器则定义根据消息头中携带的目标医院信息自动路由到目标医院所在的医院服务，最后目标医院服务的返回结果将通过通道返回到网关，从而实现系统集成的闭环。

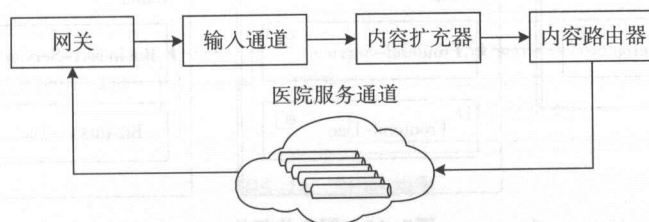


图5-11 基于企业服务总线的移动医疗系统集成实现效果图

业界基于如何实现服务总线提供了多种第三方工具，如 Mule ESB、Apache Camel 和 Spring Integration，这些工具都提供了企业级应用程序集成核心组件的主要实现模式^[8]。

5.2.3 系统扩展

所谓可扩展，扩展的是业务。当我们考虑往某个系统中添加新业务时，不需要改变原有的各个子系统而只需把新业务封闭在一个新的子系统中就能完成整体业务的升级，我们认为系统具有较好的可扩展性。

1. 系统扩展理念

对于实现可扩展性存在几种代表性的设计理念。

(1) 插件化

在 4.4.2 节中我们讨论了微内核架构模式，微内核架构模式又被称为插件模式，我们也看到了该模式在 Dubbo 中的应用。插件化思想为增量设计和开发提供了方便。可以先实现一个稳固的内核系统，然后通过插件和内核系统之间的契约，在不对核心系统进行大量修改的情况下逐渐地增加功能和特性。

对于基于产品的应用，插件化是一开始的首选。由于插件之间的低耦合，改变通常是隔离的，可以快速实现。通常架构的初期往往表现为集中式的单块系统，单块系统的内核系统是稳定且快速的，具有一定的健壮性，几乎不需要修改。

(2) 服务化

随着业务功能的不断发展以及性能、数据存储等系统瓶颈问题的出现，单块

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

系统逐渐不适合系统的维护和扩展，分布式架构应运而生。通过把系统业务进行服务化，以及完善服务治理功能，系统架构就可以如同搭建积木一样构建成高度可集成、高内聚松耦合的业务系统。如图 5-12 中我们把初始的单块系统进行了服务化改造，系统主体由 Frontend-Service 和 Business1-Service 两层服务化构成，为 Web 层提供网关和核心业务服务。如果系统需要添加新的业务，则可以添加新的 Business2 服务，原先的 Business1 服务不需要做任何变动。

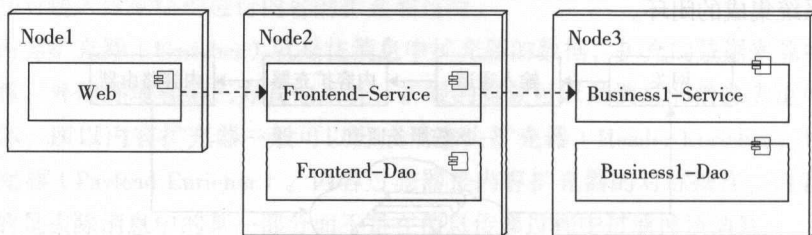


图5-12 服务化架构

（3）消息传递

当系统中分布式服务数量和种类日渐增多，而这些服务又分别属于不同业务层次时，如何合理的管理这些服务之间的调用关系，进一步确保系统的健壮性和扩展性成为系统架构设计的又一大难题。分布式服务的自身特征决定了其在时间、空间和技术上都具有一定程度的系统耦合性，在使用分布式服务时需要谨慎处理服务调用的时序、所使用的服务定义以及技术平台的差异性等问题。以各种消息传递组件为代表的中间件系统为降低系统耦合性、屏蔽技术平台差异性带来了新的思路。当不同的服务需要进行交互，但又不需要直接进行服务的定位、调用和管理时，消息中间件能显著降低系统的耦合程度，如图 5-13 中在 Business1-Service 和 Other-Service 中添加了消息传递中间件，确保两个服务在并不需要意识到对方存在的前提下进行数据的有效传输。

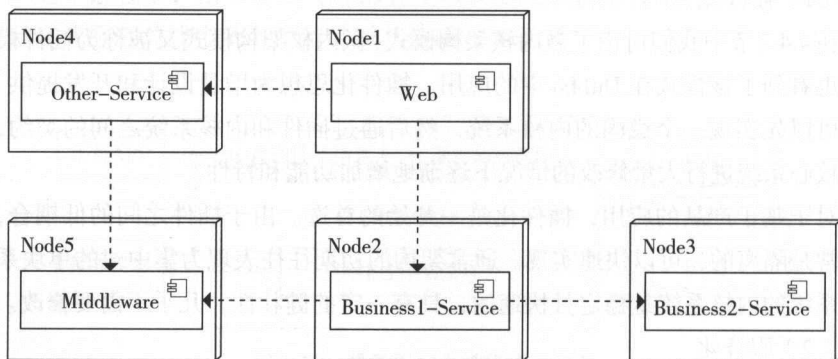


图5-13 消息中间件架构

2. 系统扩展技术

我们结合系统扩展理念中的三点思路，分别给出技术上的实现方案。

(1) 插件化

SPI 是提供给服务提供商与扩展框架功能的开发者使用的接口。SPI 约定在 META-INF/services/ 目录中创建以接口命名的文件，该文件内容为 API 具体实现类的全限定名。然后使用 ServiceLoader 类动态加载 META-INF 中的实现类，如果 SPI 的实现类为 Jar 则需要放在主程序 classpath 中。SPI 机制的结构如图 5-14 所示。



图5-14 SPI结构图

基于 SPI 机制的这一特性，我们就可以实现插件化系统，在配置时而非编译时连接类，插件化系统可以借助 SPI 实现集中化、运行时配置。

(2) 分布式服务框架

当我们把单块系统拆分成多个分布式环境下的子系统，各个子系统的业务就能够进行扩展和整合。图 5-15 是电商系统分布式环境下的业务架构图，可以看到，当我们把订单、物流、商品、库存、交易和会员等业务抽象成独立的垂直化服务，并在各个服务上层实现分布式环境下的调用和管理框架，系统的业务就可以转变为一种排列组合的构建方式。如基于订单和物流服务，我们可以构建出业务 1，业务 2 可能只依赖于交易和会员管理服务。想要达到以上效果，关键就是需要图中的分布式服务框架。分布式服务框架提供了一种按需构建的机制，在保证各个分布式服务的技术、团队、交付独立发展的前提下，确保业务整合的灵活性和高效性。

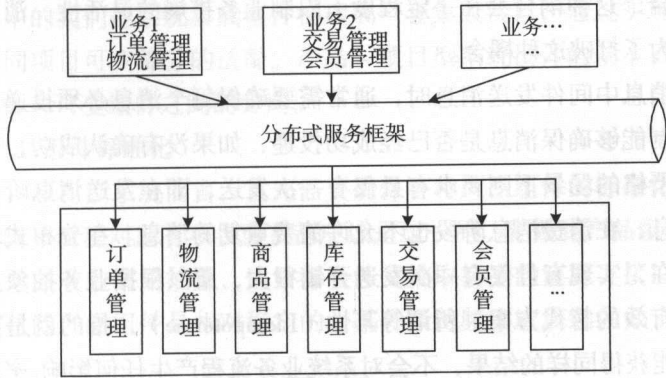


图5-15 分布式服务框架示意图

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

分布式服务框架在实现上两个主要的关注点，一个是功能需求，一个是非功能需求。提供服务接入是分布式系统的基本功能需求，包括网络通信、序列化/反序列化、传输协议和服务调用等四个基本组件。而服务路由、集群容错、服务订阅发布、服务治理和服务监控等组件更多的是为了服务调用的稳定性、可用性以及高效性。

目前，业界存在一批优秀的分布式服务框架，部分侧重于提供基本功能，如 Facebook Thirft、Twitter Finagle 和 Google gRPC，而 Alibaba Dubbo、Taobao HSF 和 Amazon Coral Service 则在实现分布式基本功能的基础上，添加了服务治理等综合性功能。这些框架在考虑服务可靠性的前提下，提供了性能优化和服务治理的实现策略。

这里以 Dubbo 为例，图 5-16 给出了其作为分布式服务框架所具有的架构设计图，关于分布式服务框架以及 Dubbo 的实现原理详细分析可参考《系统架构设计 - 程序员向架构师转型之路》^[1]。

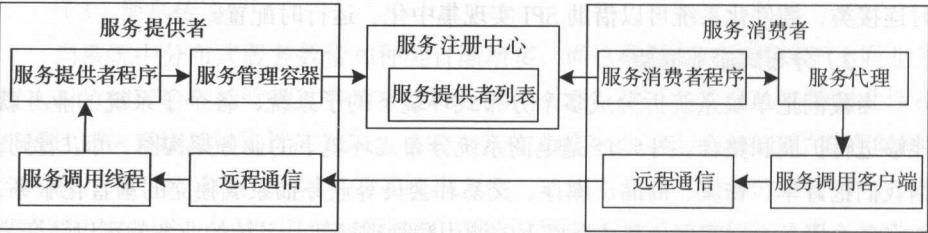


图5-16 Dubbo框架设计图

(3) 消息中间件

以 RPC 架构为实现基础的分布式服务框架，能够提供跨进程之间的方法调用，但在实现远程方法调用的同时，实际上也把方法的实现者和调用者绑定在了一起，形成一种耦合，这种耦合会在一定程度上限制业务扩展的灵活性。消息传递系统的引入就是为了打破这种耦合。

在使用消息中间件发送消息时，通常需要确保每个消息必须投递一次，通过消息确认机制能够确保消息是否已经成功投递，如果没有确认成功，可以再次发送。在更加严格的场景下则要求有且仅有一次发送，即在发送消息阶段不允许发送重复的消息，在消费消息阶段也不允许消费重复的消息。在分布式环境下由于三态性的存在，实现有且仅有一次发送开销很大，所以根据业务抽象出等幂性消息成为一种有效的替代方案。所谓等幂性 (Idempotence)，指的就是重复发送和消费消息总能获得同样的结果，不会对系统业务流程产生任何影响。

5.2.4 产品-项目适配型系统

在本节中，我们将讨论一种典型的系统开发场景，该场景以系统扩展为基本的业务架构设计需求，并综合考虑系统拆分和集成特性和实现方式。

在系统开发过程中，尤其是涉及外部系统依赖的软件开发过程中，我们会碰到类似这样的场景：团队中已经有一个产品化的软件系统，但该软件系统需要按照项目型实施，而每个项目由于面向不同客户在某一些产品组件上可能都会有其一定的独特性。这些特定的产品组件往往涉及系统集成，由于集成技术和方案的不同而需要进行二次开发，如图 5-17 所示，我们把这种场景下的系统称之为产品 - 项目适配型系统。

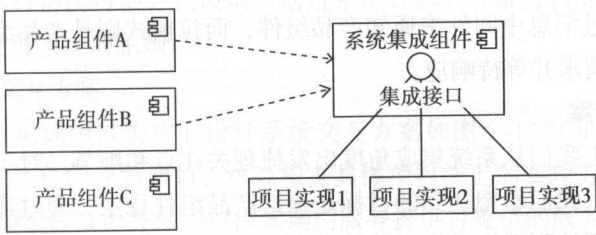


图5-17 产品-项目适配型系统

从上面的组件图中我们可以看到在产品系统中，部分产品组件（图中为产品组件 A 和 B）的运行将依赖于系统集成组件，而系统集成组件的实现将视项目而定，图中的项目实现 1、2 和 3 分别对应了三个不同客户项目的要求，在实施过程中，通过组合产品组件和不同项目线的系统集成组件实现系统集成的需求。

1. 设计理念

针对产品 - 项目型系统开发，为每个项目都保存和维护一份完整的代码显然十分低效且不符合产品化策略。通常我们都会对不同项目的系统集成部分进行抽象，如上图中的我们把系统集成组件梳理成一套集成接口，通过实现该套集成接口完成对不同项目可变部分的适配。产品 - 项目型系统设计的基本目标就是完成系统标准组件与可变组件之间的分离。

(1) 产品组件标准化

对产品 - 项目型系统而言，首先需要确保各种产品组件的标准化，即产品组件的业务逻辑和流程对所有项目而言都应该是统一的。要做到产品组件的标准化，需要对系统业务有较高的抽象，通常都会涉及抽取标准的流程、接口和数据交互格式。标准化的产品组件是相对稳定的，其升级的频率应该远远低于系统集成组件的修改频率。

（2）系统集成组件模块化

系统集成组件是可变部分，可变的是针对具体项目的系统集成方案和技术实现方法，但对产品组件而言，系统集成组件的接口和数据交互格式应该是标准的，以便和产品组件进行无缝整合。通常会提供一套默认的系统集成组件实现，如果项目上能直接使用默认实现就直接使用，如果不能就开发定制化的系统集成组件。

（3）数据推拉模式

数据推拉模式有两个维度：站在产品组件的角度和站在系统集成组件的角度。对系统集成组件而言，推模式是指具体项目中的服务提供者调用系统集成组件开放的标准服务来通知某些事件的发生，拉模式则是系统集成组件主动调用服务提供者的服务进行信息获取。而对产品组件而言，推模式是指系统集成组件在特定事件发生时通过消息中间件来通知产品组件，而拉模式则是产品组件主动向系统集成组件发起请求并等待响应。

2. 设计方案

设计方案上我们从系统集成角度出发梳理关注点和细节。对于产品-项目型系统开发而言，我们希望达到的目标是保持产品组件稳定，通过适配来自项目线的不同服务实现达到系统动态组装的效果。对产品组件而言，关注的是系统业务逻辑，本文中不展开讨论，我们来看一下系统集成组件：

（1）系统集成组件功能

系统集成组件的主要任务是完成来自项目线的服务适配，同时，作为系统中可变部分需要确保自身服务的稳定性和数据的可靠性。系统集成组件包含以下几个主要方面。

· 标准化服务定义

系统集成组件对外的服务格式必须统一才能确保与产品组件进行适配，所以首先需要确定服务定义的格式和传输方式。

· 数据验证

对服务提供者发送的请求进行删选和过滤，只有真实的业务请求数据才能经由系统集成组件流转到产品组件。

· 数据转换

数据转换包括两个方面：数据格式转换和数据内容转换。在实现数据格式转换的同时，服务提供者提供的服务中可能因为字段数量、含义等的差异性，不能完全适配到标准化服务，我们也需要对这些字段进行内容转换，确保每个字段都能符合标准化服务中的业务定义。

· 数据传递

数据传递方式上需要考虑数据的推拉模式，也可以引入回调等手段，但本质

上还是要看业务的场景和性能上的需求。系统解耦是产品组件和系统集成组件之间需要考虑的一个切入点，使用各种消息中间件技术可以提供系统的扩展性，消息异步回调也能提升性能和用户体验。

- 数据存储

系统集成组件在分层上有时候只是充当比较单纯的转化层功能，但有时候也是作为一个单独的子系统进行运作，具有本地的数据存储功能。数据存储可以是持久化的数据库存储，也可以是内存中的缓存处理。

- 系统监控

系统集成组件作为可变部分，通常都应具备高于产品组件的监控功能，因为服务提供者的运行情况可能存在波动。通过系统日志、异常分析等手段追踪问题的来源，便于发现系统的瓶颈。

（2）系统交互方案

我们以系统集成组件为中心设计系统交互方案如图 5-18 所示，该交互方案包含了系统集成组件的完整接口，可以根据具体项目线上的实现方案进行裁剪。在图 5-18 中，集成标准服务使用拉模式调用服务提供者提供的服务并完成适配，如果服务提供者支持使用推模式进行交互，那服务提供者也能调用集成标准服务进行数据推送。产品组件使用拉模式调用集成标准服务完成系统组装，同时集成标准服务也可以使用推模式与产品组件进行交互，这时通常采用基于事件处理机制的消息中间件。集成标准服务可以使用数据存储进行数据的保存。

在实际开发过程中，本节提出的产品 - 项目型适配式系统开发模式具有一定的代表性。不管采用何种叫法，其本质是系统可变部分和不可变部分的抽象和解耦，不仅仅可以应用于系统集成领域，也可以扩展到系统设计和开发的方方面面。

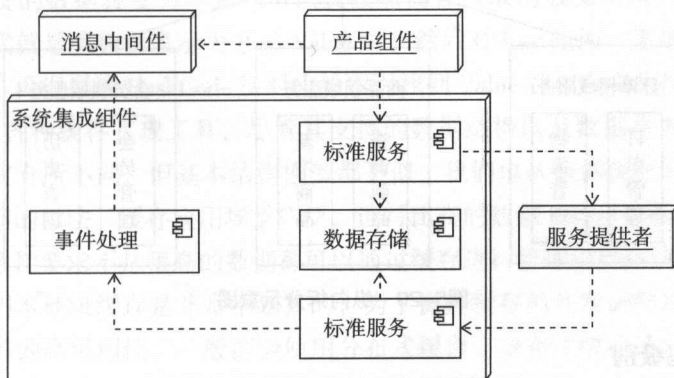


图5-18 系统交互方案

5.3 系统技术架构设计

技术架构设计的边界比较模糊，对不同受众描述的详细程度也会不同。原则上，软件设计自上而下的参与人员都会比较关注技术架构。在本节中，我们试图从系统性能、系统可用和系统安全三个方面对技术架构展开讨论。

5.3.1 系统性能

对于系统性能的处理可以采用分级思想，如图 5-19 中的描述，我们分别从应用、代码、数据和部署级别出发讨论各个级别下具体的优化方法。

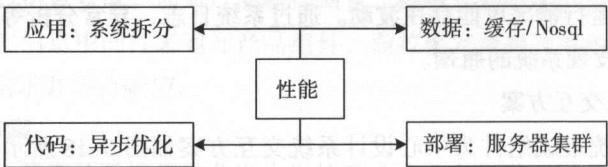


图5-19 性能优化的四个层次

1. 应用级别

在应用级别，对性能影响最大的方法仍然是拆分。这里的拆分指的是纵向拆分，也就是 5.2.1 中所介绍的拆分方法。显然，应用级别的拆分就是随着业务的增加，将一个较大的系统拆分为多个小的系统。如图 5-20 所示，对于电商行业的商品下单系统，我们可以把它拆分成订单子系统、商品子系统和会员子系统，其中每个子系统单独部署并具备自身独立的数据库实例。这样，原本在商品下单系统上的性能压力可以分摊到每个子系统中。

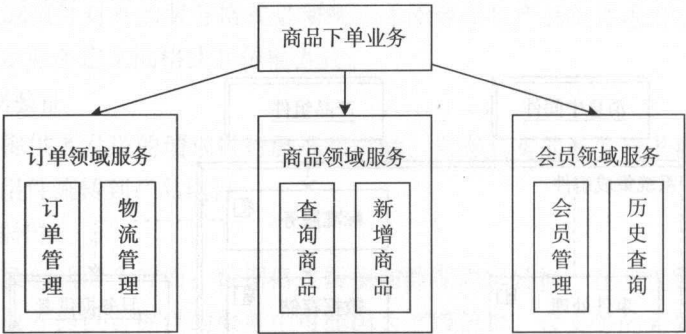


图5-20 纵向拆分示意图

2. 代码级别

现在我们已经通过拆分得到了多个独立的子系统，在每个子系统内部包含各种业务代码，对这些代码进行合理分析和优化同样能够提升性能。除了常见的多

线程技术以及在 4.4.3 中讨论的各种资源管理模式之外，我们可以采用异步化思想来满足高并发场景下的功能解耦以及性能要求。

考虑现实中的一个场景，我们去医院看病挂专家号，领取一个就诊号后一般都需要排队等待就诊，假设这位专家同时就诊的病人是 5 名，那 5 名之后的病人只能等到正在就诊的某位病人完成就诊之后才能上升一个排位。我们可以对这个场景进行抽象，挂号等待就诊的病人相当于生产者，就诊完成的病人相当于消费者，而专家就是一种共享资源。如果有两个线程 `Producer` 和 `Consumer` 分别代表生产者和消费者，它们在各自运行的时候需要访问资源队列，如果队列满了，必须要等待 `Consumer` 线程取走一部分元素才能继续进行，同时又需要保证线程安全，我们该如何有效地解决这个问题？

这个问题代表的就是典型的生产者 - 消费者模式，该模式是同步转异步思想的一种表现形式。实现该模式的基本思路是使用多线程操作，即如果资源队列满，则调用 `wait` 方法释放锁，一直等待到资源队列有空缺，然后在资源队列加入新的元素并调用 `notify/notifyAll` 方法来唤醒其他等待的线程。这种方案偏重于多线程的底层操作，对于应用程序而言，我们并不鼓励使用这种底层操作构建多线程系统，因为其复杂性远大于所带来的效果。幸好在 `JDK` 中我们有封装多线程操作的 `BlockingQueue` 组件可以实现上述需求，`BlockingQueue` 组件为我们实现生产者消费者模式提供了良好的支持。这是异步化操作的一个具体示例，代表了解决同步问题的一种思路。

3. 数据级别

数据级别提升性能的切入点包括分布式缓存和 `Nosql` 存储。

(1) 分布式缓存

缓存的作用在于减少数据的访问时间和计算时间，表现为把来自持久化或其他外部系统的数据转变为一组可以直接从内存获取的数据结构的过程。缓存的表现形式就是将数据表示为 `Key-Value` 对，然后对 `Key` 施加一定的算法获取其 `HashCode`，再根据该 `HashCode` 所对应的索引找到 `Value` 在内存中的位置并获取该 `Value` 值。各种缓存实现工具，尽管其支持的数据结构以及数据在内存中的分配和查找方式有所不同，但基本结构模型都类似，我们也认识到缓存本质上是一种时间换空间的做法。缓存应用场景广泛，读写比高的数据、热点数据、共享数据、对实时一致性要求不是很高的数据都可以通过缓存进行管理以提高访问效率。

仅使用本地级缓存是非常不划算的，为了提升缓存的并发访问效率以及确保缓存服务器的高可用性，一般都会使用分布式缓存。分布式缓存适合大量集中访问的数据，很多时候作为数据库前置组件用于挡住数据洪峰，也可作为服务器之间数据共享的存储媒介。

关于分布式缓存要明确的首要问题是各个缓存服务器之间如何进行数据的管理以便应用程序在分布式环境能找到目标缓存服务器。目前存在两种典型的分布式缓存表现形式，一种代表着所有缓存服务器通过数据同步的方式以确保应用程序的请求在任何一台缓存服务器上都能得到相同的结果，而另一种则使用完全不同的机制，即各个缓存服务器之间保持独立，应用程序通过一定的缓存交互协议能够从一而终找到特定的某一台目标缓存服务器进行数据操作。基于这两种分布式缓存的构建方式，业界也有一批优秀的实现工具可供使用，如 Memcached 和 Redis。

（2）Nosql

所谓 Nosql 一般是指那些不使用 SQL、不需要 Schema、不完整的事务支持，并且天生具备分布式特性的存储方式，其设计目标就是支持高并发的数据访问，以及提供高效的数据交互方式。

在 Nosql 中存在一个重要概念就是聚合（Aggregation），聚合一词我们已经在 5.2.1 节领域驱动思想中进行介绍，两者体现的是同一种思想，这也从另一方面证实了聚合的合理性。聚合型的 Nosql 表现模型主要有三种，分别是键值数据模型、文档数据模型和列族数据模型。键值数据模型中值可以是任意数据结构，但键一般需要专门的设计。该种数据模型适合存放会话信息、用户配置信息、基于内存计算类信息等，Memcached 和 Redis 就是典型的键值数据模型，其他如 DynamoDB、Riak 等也属于这一类型。文档数据模型中属性可以动态改变，非常适合内容管理、分析以及电子商务类系统，典型的代表就是 MongoDB。而以 HBase、Cassandra 为代表的列族数据模型则对行键设计有较高要求，适合内容管理、事件记录等应用场景。

针对性能，Nosql 所提供的基于纵向扩展和横向扩展的数据分布式模型能够在海量数据下提供很好的数据存储和访问效率。数据分布式基本模型有三种实现方式，包括复制（Replication）、分片（Sharding）以及两者之间的混合。复制可以提供主从复制和对等复制等实现方式，从而确保读写分离。而在分片机制下，要访问的数据都在同一节点上，可以实现负载均衡和自动分片。图 5-21 展示了复制和分片的基本实现方式。

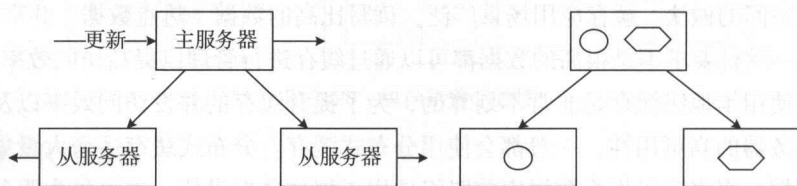


图5-21 复制与分片的基本实现方式

同时,我们也需要明确一点,Nosql 在提供强大的数据存储和获取功能的同时,其在设计理念上放弃的是关系数据库中的数据强一致性,转而提供了最终一致性。关于数据强一致性和最终一致性的讨论我们将在下一节中进行展开。

4. 部署级别

在系统部署上,首先我们可以选择一款高性能的服务器,然后通过建立集群的方式以提高并发环境下的响应性能。

(1) 高性能服务器

Nginx 是一款轻量级 Web 服务器,具有占有内存少,并发能力强,支持 Linux 和 Windows 等特点。相比轻量级、中小型和并发量不大的 Tomcat 而言,Nginx 更加适合满足复杂业务场景需求。Nginx 提供了基本 HTTP 服务,包括处理静态文件、反向代理并通过缓存进行加速、简单的负载均衡、模块过滤器功能和支持 HTTP 下安全套接层 SSL 等功能。反向代理、负载均衡和 Web 缓存可以说是 Nginx 的核心功能。

对于 Web 请求的处理方式上存在多进程和多线程两种方式。多进程方式相较于多线程方式设计和实现简单,但因为创建进程需要较大的内存复制等资源和时间开销,往往采用进程预生成技术以降低运行时创建成本。而线程开销低于进程,但共享资源导致需要任务调度和同步,实现难度大。Nginx 在请求处理机制上采用多进程方式,包括一个主进程和多个工作进程,同时结合事件驱动模型实现异步非阻塞。

Nginx 使用 IO 多路复用实现事件处理模型,IO 多路复用 in 实现方法上包括 Select 模型、Poll 模型、Epoll 模型等。Select 模型同时支持 Windows 和 Linux,分别创建所关注的 read 事件、write 事件、exception 事件的描述符集合,然后调用底层 select 函数等待事件发生,轮询事件描述符集合,如有事件发生则处理。Poll 和 Epoll 模型主要应用于 Linux 系统,都是对 Select 模型的优化。Poll 模型对 read 事件、write 事件、exception 事件创建统一事件描述符集合,而 Epoll 模型则不使用轮询进行事件通知,而是高效的等待内核上报事件。

(2) 服务器集群

集群(Cluster)指的是将几台服务器集中在一起,实现同一业务。系统部署环境中的每一个节点,都可以做集群。集群概念的提出同时考虑到了分布式系统中性能和可用性问题,一方面,集群的负载均衡机制可以将业务请求分摊到多台单机性能不一定非常出众的服务器,另一方面集群的容错机制确保当集群中的某台机器无法正常提供服务时整个集群仍然可用。关于集群的可用性我们将在下一节讨论,这里重点介绍负载均衡。

所谓负载均衡(Load Balance),简单讲就是将请求分摊到多个操作单元上进

行执行,如图 5-22 中来自客户端的请求通过负载均衡机制将被分发到各个服务器,根据分发策略的不同将产生该策略下对应的分发结果。分发策略在软件负载均衡中的实现体现为一组分发算法,通常称为负载均衡算法。负载均衡算法可以分成两大类,即静态负载均衡算法和动态负载均衡算法。

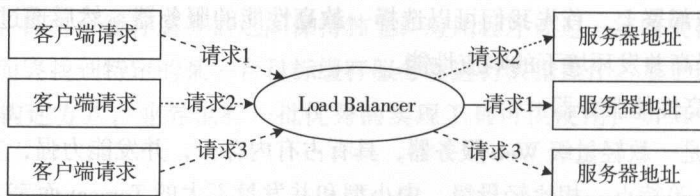


图5-22 负载均衡示意图

· 静态负载均衡算法

静态负载均衡算法的代表是各种随机（Random）和轮询（Round Robin）算法。采用随机算法进行负载均衡在集群中相对比较平均。随机算法实现也比较简单,使用 JDK 自带的 Random 相关随机算法就可指定服务提供者地址。随机算法的一种改进是加权随机（Weight Random）算法,在集群中可能存在部分性能较优服务器,为了使这些服务器响应更多请求,就可以通过加权随机算法提升这些服务器的权重。加权轮循（Weighted Round Robin）算法同样按照权重,顺序循环遍历服务提供者列表,到达上限之后重新归零,继续顺序循环直到指定某一台服务器作为服务的提供者。普通的轮询算法实际上就是权重为 1 的加权轮循算法。

· 动态负载均衡算法

所有涉及权重的静态算法都可以转变为动态算法,因为权重可以在运行过程中动态更新。例如动态轮询算法中权重值基于对各个服务器的持续监控并不断更新。基于服务器的实时性能分析分配连接（比如每个节点的当前连接数或者节点的最快响应时间）是常见的动态策略。类似的动态算法还包括最少连接数（Least Connection）算法和服务调用时延（Service Invoke Delay）算法,前者对传入的请求根据每台服务器当前所打开的连接数来分配;后者中服务消费者缓存并计算所有服务提供者的服务调用时延,根据服务调用和平均时延的差值动态调整权重。

5.3.2 系统可用

前一节中我们已经提到集群同时可以用于确保系统性能和系统可用性,但除了集群机制,系统可用上我们还可以使用服务分级、降级和隔离措施。

1. 对等集群

单点失效（Single Point Of Failure, SPOF）是指一个系统中的某个部件,如果它失效或停止运转,将会导致整个系统不能工作。解决 SPOF 的基本原理是对所

有的 SPOF 使用冗余复制。系统级冗余的实现是使用备用服务器承担失效服务器的工作，也就是前面所说的集群。

对于用于确保可用性的集群而言，无状态至关重要。在系统运行中不改变对象状态，避免了状态管理的复杂度。而对于集群，无状态性所带来的优势更为明显，通过简单的添加服务器就可以确保性能和可用性的提升。无状态的集群中所有的服务器都提供同样的服务，客户端只要连接一个服务器完成服务调用即可，任何一台服务器宕机都不影响客户端正常使用，也就是说无状态服务的失效转移成本几乎为零。这样的集群通常称为对等集群（Peer-to-peer Server Cluster）。图 5-23 就是一个对等集群示例，展示的是 Dubbo 中使用 Zookeeper 对等集群作为注册中心进行服务发布和推送的场景。

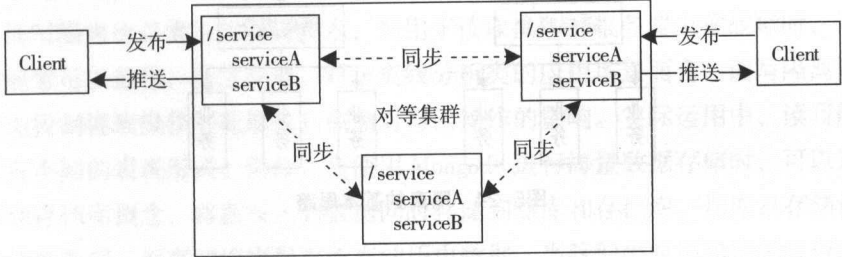


图5-23 对等集群示例

2. 集群容错

当服务运行在一个集群中，出现通信链路故障、服务端超时以及业务异常等场景都会导致服务调用失败。容错(Fault Tolerance)机制的基本思想是重试和冗余，即当一个服务器出现问题时不妨试试其他服务器。集群的建立已经满足冗余的条件，而围绕如何进行重试就产生了几种常见的集群容错策略。

首先最常见的是 Failover，Failover 即失效转移，当发生服务调用异常时，重新在集群中查找下一个可用的服务提供者。为了防止无限重试，通常对失败重试最大次数进行限制；Failback 可以理解为失败通知，当服务调用失败直接将远程调用异常通知给消费者，由消费者捕获异常进行后续处理；Failsafe（失败安全）策略中，当获取服务调用异常时，直接忽略。通常用于写入审计日志等操作，确保后续可以根据日志记录找到引起异常的原因并解决。该策略可以理解作为一种简单的熔断机制（Circuit Breaker），为了调用链路的完整性，在非关键环节中允许出现错误而不中断整个调用链路；最后是 Failfast（快速失败）策略，该机制在获取服务调用异常时，立即报错。显然，Failfast 已经彻底放弃了重试机制，等同于没有容错。在特定场景中可以使用该策略确保非核心业务服务只调用一次，为重要的核心服务节约宝贵时间。

3. 服务隔离

服务隔离上包括一些常见的隔离思路以及特定的隔离实现技术框架。

(1) 常见隔离思路

所谓隔离（Isolation）本质上是对系统或资源进行分割，从而实现当系统发生故障时能限定传播范围和影响范围，即发生故障后只有出问题的服务不可用，保证其他服务仍然可用。隔离的基本思路如图 5-24 所示。

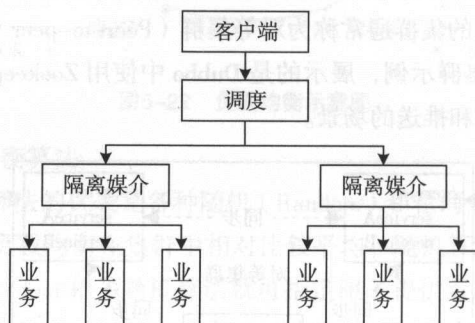


图5-24 隔离的基本思路

· 线程隔离

线程隔离主要通过线程池（Thread Pool）进行隔离，在实际使用时我们会把业务进行分类并交给不同的线程池处理，当某个线程池处理一种业务请求发生问题时，不会将故障扩散到其他线程池，也就不会影响到其他线程池中所运行的业务，从而保证其他服务可用。如图 5-24 所示，我们可以把隔离媒介替换成线程池就能起到线程隔离的效果。

· 进程隔离

进程隔离比较好理解，就是将系统拆分为多个子系统来实现物理隔离，各个子系统运行在独立的容器和 JVM 中，通过进程隔离使得某一个子系统出现问题不会影响到其他子系统。图 5-24 中，我们同样可以把隔离媒介简单替换成 JVM 就能起到进程隔离的效果。

· 集群隔离

集群隔离是进程隔离的升级版。当将某些服务部署成集群，并对服务进行分组集群管理时，某一个集群出现问题之后就不会影响到其他集群，从而实现了故障隔离。我们把图 5-24 进行简单改造就能得到如图 5-25 所示的集群隔离效果图，该图中包括了单一的服务 A 集群，也包括分组的服务 B 集群。

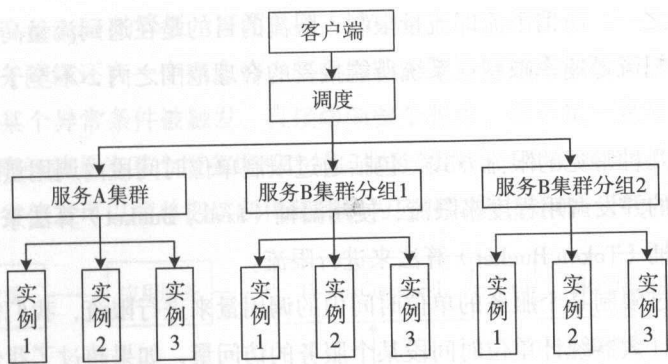


图5-25 集群隔离示意图

· 读写隔离

读写隔离也是常见的隔离技术，当用于读取操作的服务器出现故障时，写服务器照常可以运作，反之亦然。对于离线分析类的应用场景而言，读写隔离可以很好地控制读取操作可能形成的瓶颈对写入操作的影响。实际运用中，读写隔离可以有不同的表现形式。例如，当使用 MongoDB 进行海量数据存储时，可以采用热库和存档库概念，将新写入的数据同时存储到热库和存档库，热库只存储最近一个月的数据，而存档库则保存所有的历史数据。当我们想对历史数据进行离线处理时，可以确保热库不受影响。

(2) Hystrix 实现隔离

Hystrix 是 Netflix 开源的一款针对分布式系统的容错库，可用来隔离分布式服务故障，确保系统可用性。Hystrix 组件提供了两种隔离的解决方案，即线程池隔离和信号量隔离，两种隔离方式都是限制对共享资源的并发访问量。以线程隔离为例，Hystrix 使用命令模式 HystrixCommand (Command) 包装依赖调用逻辑，每个命令在单独线程中执行。同时，为每个依赖提供一个小的线程池，如果线程池已满调用将被立即拒绝。

结合上文中介绍的服务和服务分组概念，Hystrix 可以粗粒度实现隔离，也可以细粒度实现隔离，包括服务分组 + 线程池、服务分组 + 服务 + 线程池和混合实现三种模式。其中服务分组 + 线程池模式是一种粗粒度实现，一个服务分组配置一个隔离线程池即可；而服务分组 + 服务 + 线程池则是一种细粒度实现，一个服务分组中的每一个服务配置一个隔离线程池；在混合实现模式下，一个服务分组配置一个隔离线程池，然后对重要服务单独设置隔离线程池。

4. 服务限流

为了保证在业务高峰期线上系统也能保证一定的弹性和稳定性，限流是最常

采用的方案之一。所谓限流即流量限制，限流的目的是在遇到流量高峰期或者流量突增时，把流量速率限制在系统所能接受的合理范围之内，不至于让系统被高流量击垮。

目前几种常见的限流方式，包括通过限制单位时间段内调用量来限流、通过限制系统的并发调用程度来限流、使用漏桶（Leaky Bucket）算法来进行限流以及使用令牌桶（Token Bucket）算法来进行限流。

对于通过限制某个服务的单位时间内的调用量来进行限流，我们需要做的就是通过一个计数器统计单位时间段某个服务的访问量，如果超过了我们设定的阈值，则该单位时间段内就不允许继续访问，或者把接下来的请求放入队列中等待到下一个单位时间段继续访问。这里，计数器需要在进入下一个单位时间段时先清零。

对于通过并发限制来限流，我们通过严格限制某服务的并发访问程度，其实也就限制了该服务单位时间段内的访问量，而且这与第一种限流方案相比，它有着更严格的限制边界，因为如果采用第一种限流方案，如果大量服务调用在极短的时间内产生，仍可能压垮系统。

5. 服务降级

服务降级指的是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务有策略的降级，以此释放服务器资源以保证核心任务的正常运行。

（1）服务降级的理念

降级可以是有计划的执行，也可以是被动触发。如电商网站在双十一期间对部分非核心业务进行手工降级就属于前者，而后者则包括系统运行时可能出现的异常情况，为了控制异常的影响范围可以实现自动服务降级。

服务降级在实现上一般需要对业务有层级之分，也就是需要服务分级。对每个服务进行等级管理之后，降级一般是从最外围、最低级的服务开始。

（2）Hystrix 实现降级

服务降级在概念上比较容易理解，最终让用户体验到的是某些功能暂时不可达或不可用。实现上也有一些技术体系，这里我们仍然使用 Hystrix 来介绍其提供的服务降级机制。

在 Hystrix 中，HystrixCommand 是重中之重，在 Hystrix 的整个机制中，涉及依赖边界的地方，都是通过这个 Command 模式进行调用的，显然这个 Command 也负责了核心的服务降级处理。其子类要实现的方法主要有两个。一个是 run 方法，用于实现依赖的业务逻辑；另一个就是 getFallBack 方法，实现服务降级处理逻辑。

基于 HystrixCommand 的降级示意图见图 5-26。

关于服务降级还有一个概念称为服务熔断，服务熔断类似现实世界中的“保险丝”，当某个异常条件被触发，直接熔断整个服务，而不是一直等到此服务超时。而服务降级就是当某个服务熔断之后，服务端准备一个本地的 fallback 回调，返回一个缺省值。可以简单把熔断机制理解为应对降级目标的一种实现。

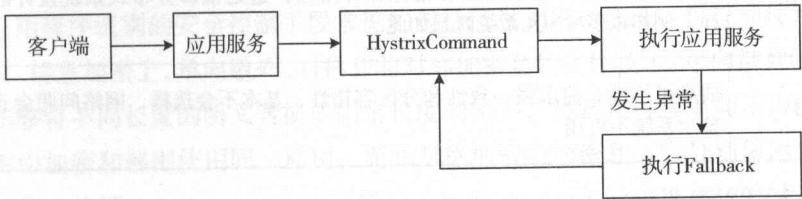


图5-26 HystrixCommand实现降级

6. 数据一致性

(1) CAP 理论

CAP 理论^[20]指的是在一个分布式系统中，无法同时实现一致性(Consistency)、可用性(Availability)和分区容错性(Partition tolerance)。试想在图 5-27 所示的分布式环境中，一致性(C)是指在分布式系统中的所有数据备份在同一时刻是否拥有同样的值；可用性(A)是指在集群中一部分节点故障后，集群整体是否还能正常响应请求；分区容错性(P)中的分区相当于对通信的时限要求，系统如果不能在一定时限内达成数据一致性，就意味着发生了分区的情况，也就是说整个分布式系统不再互联。

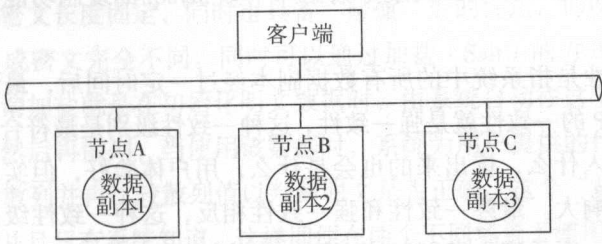


图5-27 典型的分布式系统

现实环境中，出现分区，就相当于分布式环境中的节点分成了两组，彼此之间无法通信。这种情况比其中一组全宕机更复杂，因为两组都可能独立对外提供服务。由于当前的网络硬件肯定会出现延迟丢包等通信异常问题，所以一般认为分区容错性必须实现。关于一致性、可用性和分区容错性三种之间的排列组合参考表 5-2。

表5-2 CAP组合

组合维度	组合描述
CA	放弃分区容错性，加强一致性和可用性，即是传统的单机数据库的选择
AP	放弃强一致性，追求分区容错性和可用性，这是很多分布式系统设计时的选择，例如很多NoSQL系统就是如此
CP	放弃可用性，追求强一致性和分区容错性，基本不会选择，网络问题会直接让整个系统不可用

(2) BASE 思想

eBay 的架构师 Dan Pritchett 源于对大规模分布式系统的实践总结，在 ACM 上发表文章提出 BASE 理论^[21]，BASE 理论是对 CAP 理论的延伸，核心思想是即使无法做到强一致性，应用仍然可以采用适合的方式达到最终一致性。BASE 是指基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventual Consistency）。

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。例如，我们在前文中提到的服务限流和服务降级都是基本可用思想的具体体现。

软状态是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。分布式存储中一份数据一般都会有若干个副本，允许不同节点间副本同步的延时就是软状态的体现。关系型数据库中如 Mysql 的复制功能也是该思想的一种体现。

最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。CAP 的一致性就是强一致性，这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响大。最终一致性和强一致性相反，这种一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态。

(3) 一致性与可用性

通过对 CAP 理论与 BASE 思想的分析，我们明白一致性和可用性之间是一种平衡关系，需要根据具体场景进行权衡。为了达到系统可用性，分布式往往会牺牲一部分强一致性转而提供最终一致性。

5.3.3 系统安全

1. 安全性概述

我们的系统可能面临着各种潜在的网络攻击手段，包括 XSS 攻击、注入攻击、CSRF 攻击和 DDoS 攻击等，应对攻击的基本切入点有两种，即网络架构和应用程序。本书主要关注应用程序级别的安全性实现。

应用程序级别的安全控制手段常见的包括信息加密、认证、授权和使用安全性协议。信息加密上，单向散列、对称和非对称加密是代表性技术。单向散列（Hash）加密能够将不同长度的明文转换成固定长度的密文，且转化过程单向不可逆；对称加密中加密和解密使用同一密钥，而非对称加密中加密和解密分别使用公钥和私钥机制。认证（Authentication）和授权（Authorization）的含义比较容易混淆，认证的目的在于回答“你是谁”这一个问题，常见的包括摘要认证和签名认证，而授权在于明确的“你是谁”之后进一步明确“你能做什么”，通用的授权模型是对资源、权限、角色和用户的组合。在协议层面也有一些成熟方案，如 HTTPS（HTTP over Secure Socket Layer）协议能将整个通信过程加密，而 OAuths 协议则通过分布式环境下开放和消费第三方接口解决授权问题。

2. 安全性实现技术

常见的安全性实现技术包括加密、认证和协议。

（1）加密算法

· 单向散列加密

单向散列加密算法常用于生成消息摘要（Message Digest），其主要特点在于单向不可逆和密文长度固定，同时也具备“碰撞”少的优点，即明文的微小差异就会导致所生成密文完全不同。同时可以通过加盐（Salt）的方式进一步提高破解的难度。所谓加盐就是在初始化明文数据时，由系统自动往这个明文里添加一些附加数据，然后再散列，当使用该数据时，系统为用户提供的代码撒上同样的“盐”，然后散列并再比较散列值以判断明文是否正确。这个“盐”值是由系统随机生成的，并且只有系统知道。这样即便在两个不同场景下使用了同样的明文，由于系统每次生成的盐值不同，它们的散列值也不同。常见的单向散列加密算法实现包括 MD5 和 SHA。

· 对称加密

对称加密（Symmetric Encryption）中加密和解密采用统一算法，密钥对称。使用对称密钥的优点在于简单、高效、长密钥难破解，但需要确保密钥交换过程的安全性。DES 和 AES 算法是目前对称加密的主要实现方式。

· 非对称加密

与对称加密在加密和解密时使用同一个密钥不同，非对称加密（Asymmetric Encryption）需要两个密钥来进行加密和解密，这两个密钥分别称为公钥（Public Key）和私钥（Private Key）。如图 5-28 所示，首先乙方生成一对密钥（公钥和私钥）并将公钥向甲方公开，得到该公钥的甲方使用该密钥对机密信息进行加密后再发送给乙方，乙方再用自己保存的私钥对加密后的信息进行解密。在传输过程中，即使攻击者截获了传输的密文，并得到了乙的公钥，也无法破解密文，因为只有乙的私钥才能解密密文。同样，如果乙要回复加密信息给甲，那么需要甲先公布自己的公钥给乙用于加密，甲自己保存甲的私钥用于解密，甲乙之间使用非对称加密的方式完成敏感信息的安全传输。显然非对称加密实现和密钥管理比较复杂，目前典型的实现有 RSA 算法。

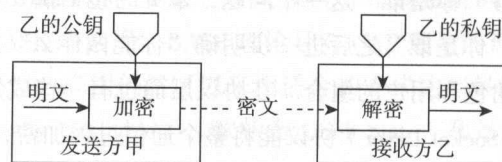


图5-28 非对称加密过程示意图

以上三种加密算法各有其应用场景，单向散列加密主要用于生成信息摘要和随机数，对称加密用于通信加密，而非对称加密用于信息安全传输。同时，算法之间也可以混合使用，如可以使用非对称加密传输对称密钥，使用对称加密进行数据加解密。

（2）认证

加密算法是一种基础设施，可以基于各种加密算法完成特定场景下的业务需求，比如认证。Web 应用中 HTTP 请求实际上比较脆弱，抓包工具也很强大，开放环境下对请求和响应进行认证重要性不言而喻。常见的认证方式有摘要认证和签名认证。

· 摘要认证

摘要认证中摘要的对象是客户端参数和服务端响应，即在请求-应答式交互过程中，需要站在发生交互的客户端和服务端分别判断对方是否合法。摘要认证的过程也比较简单，即通过 MD5 或 SHA 算法结合加盐机制进行摘要的生成和比对以判断信息是否被篡改。整个认证就是一个客户端参数摘要生成→服务端参数摘要验证→服务端响应摘要生成→客户端响应摘要验证的闭环交互过程。这里需要考虑如何保证客户端和服务端使用同一个 Salt，服务端可以在认证之前通过某种方式把 Salt 发给需要接入的客户端。

· 签名认证

摘要认证的主要问题就是如何防止 Salt 泄露, 而签名认证则不使用 Salt, 基本思路是使用非对称加密算法加密数字摘要, 是混合算法的一种具体应用。同摘要认证一样, 签名认证也通过客户端参数签名生成→服务端参数签名验证→服务端响应签名生成→客户端响应签名验证完成闭环。摘要认证中通常使用 MD5withRSA 和 SHA1withRSA 等算法组合。

(3) 协议

· HTTPS

摘要认证和签名认证能够对信息的发送方和接收方的身份进行有效验证, 但敏感信息仍然以明文方式在网络上传递。HTTPS 协议通过对整个通信过程加密的方式确保敏感信息不被泄漏, 可以把 HTTPS (HTTP over SSL) 看作是 HTTP 协议加上 SSL/TLS 的结合体。

SSL/TLS 协议的基本思路是采用公钥加密法, 也就是说, 客户端先向服务器端索要公钥, 然后用公钥加密信息, 服务器收到密文后, 用自己的私钥解密。因为需要对整个通道的所有信息进行加密, 而非对称加密计算量太大导致加密效率过低, SSL/TLS 协议的做法是针对每一次会话 (Session), 客户端和服务端都生成一个会话密钥 (Session Key), 用它来加密信息。由于会话密钥是对称加密, 所以运算速度非常快, 而服务器公钥只用于加密会话密钥本身, 这样就减少了加密运算的消耗时间。这样 SSL/TLS 协议的基本过程就分成三步, 即客户端向服务器端索要并验证公钥→双方协商生成会话密钥→双方采用会话密钥进行加密通信。其中前两步, 又被称为握手阶段 (Handshake)。

SSL/TLS 协议的握手阶段非常复杂, JSSE (Java Secure Socket Extension, Java 安全套接字扩展) 是 SSL 和 TLS 的纯 Java 实现, 通过它可以透明地提供数据加密、服务器认证、信息完整性等功能, 如同使用普通的套接字一样使用安全套接字。通过 JSSE, 开发者很轻松将 SSL 协议整合到应用程序中, 并且 JSSE 能将安全隐患降到最低点。

· OAuth

OAuth 是 Open Authorization 的简称, 面向 SOA 和 ESB 等分布式场景下开放和消费第三方接口, 用户平台商保障用户私有数据合法访问。OAuth 协议解决的是授权问题而不是认证问题。在应用过程中, 不需要账户信息访问第三方应用, 平台商通过 OAuth 协议对第三方应用进行授权, 从而使第三方应用、用户和授权平台形成一个整体。

OAuth 协议在实现上体系比较复杂, 综合应用摘要认证、签名认证、HTTPS 等安全性手段, 需要提供 Token 生成和校验、分布式 Session 和公私钥管理等功

能，同时需要开发者入驻并进行权限粒度控制。一般我们可以借助 Apache Oltu、Spring Security for OAuth 等工具来搭建 OAuth 平台。

5.4 本章小结

系统架构设计存在分层、分割、分布式、集群、缓存、异步、安全等常见的设计方法。本章首先对这些常见的设计方法从架构设计的层次和维度对其进行分类，认为任何的架构设计方法都可以归为两大类型，即业务架构设计和技术架构设计。

业务架构设计从业务发展的需要出发，包括系统拆分、系统集成、系统集成等内容，本章对这些维度从思路方法、设计策略和实现技术等几个主要方面展开讨论，并给出一些示例；而技术架构设计则更加偏重于技术本身，关注于系统性能、系统可用、系统安全等维度，本章对这些维度的介绍也是包括设计理念和相关实现技术两个方面。

6 技术创新

所谓创新是指以现有的知识和物质，在特定的环境中，改进或创造新的事物，并能获得一定有益效果的行为，这里的事物包括但不限于各种方法、元素、路径、环境等。创新包含的范围非常广，其中就包括技术创新。

什么是技术创新（Technical Innovation）？技术创新是指生产技术的创新，包括开发新技术，或者将已有的技术进行应用创新。技术创新和产品创新同属于科技创新，而正如我们上一章 5.1.1 中所讨论的业务架构和技术架构的关系一样，技术创新和产品创新有密切关系，又有所区别。对软件行业而言，通常产品的创新也伴随着技术的创新，反之也是一样。

如何实现技术创新？可以参考图 6-1 中的创新四象限理论。从图 6-1 中我们可以看到，技术创新的思路主要包括以下三个方面：

- 对新的问题使用新的方法

这是最具有创造力的一种创新模式，通过对新事物的分析将其转变为新的需求，然后引入新的技术实现该需求。

- 对新的问题使用老的方法

这应该是最常见的一种创新模式，我们使用的还是老方法，如果该老方法同样适用于新问题，那也能够带来创新的价值。

- 对老的问题使用新的方法

这种创新更多可以认为是一种改进过程，通过引入新的方法来提升原有问题的解决效率。

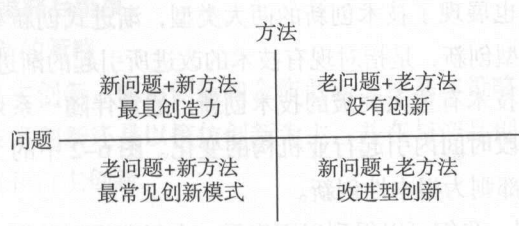


图6-1 创新的四象限理论

创新的四象限理论给我们提供了技术创新的方向，对于软件开发而言，以上三种创新思路都适合于日常的产品和技术开发过程。

6.1 技术创新概述

如同软件开发中的产品、项目、技术等我们日常都能涉及的工作一样，技术创新同样是一项实实在在的活动，需要符合技术发展和变革的客观规律，也需要梳理创新的思维和策略，并付诸具体过程与实践。

6.1.1 技术变革的基本规律

不可否认，任何产业以及产业下的产品都有其生命周期，这点对于软件行业尤为明显。随着互联网行业的蓬勃发展，很多传统行业也都纷纷转型开发出很多“互联网+”型产品。结合我们在第2章中对行业的分析，我们就会发现从行业成长的角度来研究创新过程，分析技术创新与行业成长的关系，是技术创新首先要明确的内容。这方面具有代表性的模型是A-U模型^[22]。

A-U模型认为企业的产品创新和技术创新是相互联系的，在行业成长的不同阶段，对两者的侧重有所不同，企业的创新频率和创新类型取决于行业成长的不同阶段，并把行业分为三个不同阶段，即流动阶段、过渡阶段和特性阶段（见图6-2）。

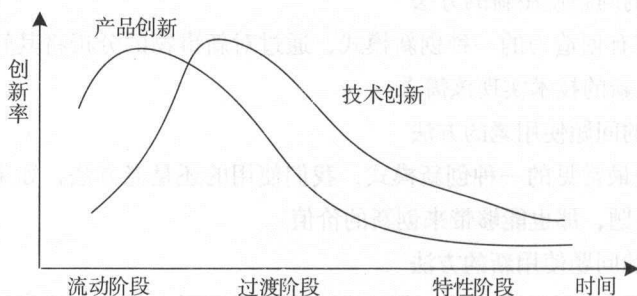


图6-2 A-U模型

图6-2中同时也展现了技术创新的两大类型，渐进式创新和根本性创新。渐进式创新就是改进型创新，是指对现有技术的改进所引起的渐进式、连续的创新。而根本性创新是指技术有重大突破的技术创新，通常伴随一系列渐进式的产品和技术创新，并在一段时间内引起行业机构的变化。图6-2中的上升曲线就是代表渐进式创新，而顶部则为根本性创新。

通过A-U模型，我们可以得到以下启示：在行业发展前期，创新的焦点集中在产品创新，竞争点在于产品的性能和功能，企业试图通过产品新颖性和功能来

占据市场；在行业趋于稳定之后，创新类型转向技术创新，竞争转向了价格和成本，企业试图通过低成本优势来赢得市场，最后，创新集中在渐进性的产品创新和技术创新，企业借此赢得竞争优势。

技术创新意味着变化，围绕 A-U 模型，我们再来看一下技术演化的过程。任何技术的发展都表现为一种 S 曲线。在图 6-3 中，我们可以看到当对某项技术持续投入时，技术所产生的效果会达到一种上限；如果我们通过技术创新的方式采用新技术替换老技术，同样，新技术也会面临另一种上限。S 曲线告诉我们技术演进的过程并不是一个无限的过程，同样具备生命周期。

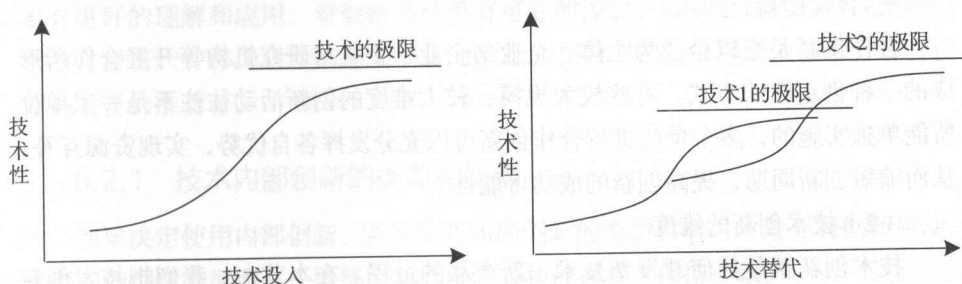


图6-3 S曲线

技术演化的生命周期表现为一种技术扩散曲线，我们已经在 2.1.3 节图 2-13 中看到过这条曲线。从本质上讲，技术扩散的核心是采用者对技术创新的学习和模仿行为，是技术创新的延伸化。技术扩散一般起始于某项根本性创新的首次商业化应用，经过推广使用，直到被淘汰，这个过程对应于一个技术周期。技术演化的过程实际上就是技术逐步扩散的过程。

6.1.2 技术创新策略与模式

根据技术扩散曲线，我们希望自己能够成为创新者，如果由于各种条件所限无法成为创新者，那也应该争取成为早期采用者。为此，技术创新需要有相应的策略和过程来支持这一目标。

1. 技术创新策略与维度

(1) 技术创新的策略

技术创新有自主创新、模仿创新和合作创新三种基本策略。对于软件开发而言，目前大多数技术创新还是以模仿创新为主，并在与产品创新相结合的特定领域中实现合作创新和自主创新。

· 自主创新

所谓自主创新是指企业主要依靠自身的技术能力进行研究开发，并在此基础

上实现研发成果的产品化，从而获取市场的认可。自主创新具有根本性创新的特质，因为一种新技术的率先创新者一般只有一家，而其他采用该技术的都是模仿者。

· 模仿创新

模仿创新是指在其他企业进行率先创新的影响下，通过合法手段引入新技术，并在原有技术体系的基础上进行改进的一种创新形式。模仿创新并不是简单的复制，而需要有所改进。通常，在实施模仿创新时，先进行高质量的模仿，然后再找到某些切入点进行改进。

· 合作创新

合作创新是指以企业为主体，企业与企业、企业与研究机构等开展合作所形成的一种创新组织方式。有些较大规模、较大难度的创新活动往往不是一家单位所能单独实施的，多个单位进行合作创新可以充分发挥各自优势，实现资源互补，从而缩短创新周期，提高创新的成功可能性。

（2）技术创新的维度

技术创新就是如何开发新技术、新产品的过程。在本节中，我们把技术创新分成两个维度，即内部创新和外部创新。顾名思义，内部创新的力量来自于团队内部，而外部创新则从外部资源汇总获取新的技术。

技术团队可以采用其中的任意一种，也可以把两者结合起来一起实施。一般新技术的开发需要企业级别的支持以及各个部门之间的高度协作，同时，使用新技术需要时间，如果市场上已有竞争产品即将投放市场，那么使用内部创新就会处于劣势。相反，如果向已经存在或即将进入市场的公司购买技术，就可以迅速获取新技术，从而快速推出新产品。

2. 技术创新过程模式

技术创新是一个将知识和技能转化为产品的过程，这个过程开展的方式视行业和企业而异，业界具有代表性的技术创新过程模式包括：

（1）需求驱动创新

绝大多数的技术创新都应该来自于业务需求的驱动。行业的扩展、产品的演进、用户群体的变迁都会刺激企业进行技术创新。在需求驱动的创新模式下，市场是研究技术开发构思的源头，业务需求为产品和技术创新提供了新的机会，也就要求技术做出相应的变化。

（2）技术驱动创新

对于某些产品研发而言，其业务需求相对保持稳定且简单，变化的更多是用户体验层面的内容。这时候，研究和开发能够提升用户体验的新的技术就是一种典型的创新性行为。技术创新表现为由技术反哺产品的一种线性过程。很多根本

性创新来自于技术驱动类创新，对新的技术上的发现能刺激技术人员为之寻找新的应用领域。

6.2 内部创新

关于内部创新首要的问题在于是否需要创新，这是包括技术管理者在内的很多人需要考虑的问题。如果决定要创新，是使用内部创新还是下一节要介绍的外部创新也是一个问题。内部创新有其固有的优势，表现在对创新过程中使用的技术有更好的理解 and 应用、对创新及结果有更好的控制、能够更加容易形成组织级别的过程资产并作为后续渐进式创新的基础。在内部创新过程中，技术管理者往往是主要的推动者。

6.2.1 技术内部创新的类型和要素

如果决定使用内部创新，首先需要明确创新的类型，然后才是开展创新活动。对不同的创新类型而言，内部创新所需活动和过程基本保持一致。

1. 内部创新的类型

这里有必要强调一下，所谓的技术创新实际上不可能脱离一个具体的产品或组织，纯粹为了技术创新而技术创新是没有任何意义的，所以对于技术创新而言，除了技术本身之外，其切入点往往还包括产品或管理。

（1）面向基础研究的技术创新

基础性的研究工作往往指纯粹的技术开发，这类技术创新活动一般比较独立，并不一定有其能够承载的具体产品表现形式，或者说技术本身就是一种产品。基础性研究风险很高，但如果一旦成功，则能够很好地反哺产品。这类技术创新难度较高，在企业中相对也比较少见。

（2）面向产品的技术创新

面向产品的技术创新包括以下几种表现形式：技术应用、技术演变和产品集成。

技术应用关注于使用新的技术去解决现有问题。新技术的应用如果能够改变产品的表现形态、用户体验、运营模式等方面，并产生新的价值，那么技术应用就能成为技术创新的手段，而且是相对比较简单的手段。

技术演进则关注于现有技术的改进。随着业务的不断发展，技术发展到一定阶段势必会形成一种瓶颈，如何打破这种瓶颈并成为新的技术生产力就是技术演进在创新过程中的表现形式。

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

产品集成本质上是一种产品改进的方式，通过各个现有产品的整合和优化，能够带来功能上的改良或为已有的产品开拓新的市场。这种产品集成方式属于风险比较低的技术创新，相对而言，所能产生的价值也较低。

（3）面向过程的技术创新

面向过程的技术创新关注于结构重组和组织再造，属于面向管理的创新方式。该部分内容我们将在第8章研发过程体系建设中详细讨论如何进行过程的裁剪和创新。

2. 内部创新的要素

内部创新在实施过程中需要考虑以下几个要素：

（1）时机选择

对于内部创新而言，时机选择很重要。一方面，技术创新工作具有时效性，对于一个行业而言，技术管理者需要有一定的敏感度了解当前行业的需求以及技术创新的必要性。另一方面，对于一个组织而言，技术创新工作尽管拥有较高优先级，但仍然可能与其他战略目标相冲突。只有确定时间，才能开展后续的具体工作。

（2）资源规划

资源规划主要涉及人员和工具。对于软件开发而言，技术创新的主要投入成本是人员，一般都会使用业界主流的工具。对于人员，根据需要可能还涉及培训、招聘等工作，这些都是内部创新所需要考虑的问题。

（3）现有设施利用

技术创新并不是从无到有的过程，相反，我们可以利用现有的技术平台作为节约资源的一种手段。对于现有的技术体系，一个好的组织应该具备过程资产建设的能力，并把技术创新所产生的结果作为核心过程资产进行管理。技术创新往往是站在现有设施的基础上所开展的一系列活动。

6.2.2 技术应用创新案例

在本节中我们将通过一个技术应用创新的案例来介绍如何进行内部创新，该案例属于自主创新的一种表现形式。

1. 医疗信息系统创新

在本书2.2.3节中，我们结合互联网+的背景，通过解决方案的设计思路给出了移动医疗的一种解决方案示例，本节将在该解决方案示例的基础上分析如何进行技术创新。

（1）企业级应用与互联网应用的区别

传统医疗行业的系统基本都属于企业级应用，这些系统一般都部署在医院内

网环境，结构复杂同时有较强的安全性考虑，而互联网应用则在很多特点上与企业级应用存在较大差别。两者之间的典型对比参考表 6-1。

表6-1 企业级应用与互联网应用对比

对比维度	企业级应用	互联网应用
行业领域	区分行业，各自领域业务背景不一样，并形成了一定的门槛	跨行业，按应用类型区分
业务逻辑	业务逻辑复杂，涉及大量的数据和多人协同处理	业务逻辑简单，大部分是通过页面进行数据的增删改查
数据一致性	强调数据一致性，需要通过事务和同步等机制来保证数据的一致性	要求一致性，但为了满足高并发会关注最终一致性而不是实时一致性
并发量	不是特别大，通常在几百的并发量之内	强调高并发，支持用户数量多，并采取企业开发中极少采用的技术
开发过程	强调软件过程和行业经验，需要撰写大量的文档和多人的协同，需要版本控制和问题跟踪回溯	强调敏捷，快速开发，侧重于人员沟通和团队协作，不太重视过程和文档

通过以上对比，我们不难发现，如果我们想把原本是企业级应用的医疗系统转变为面向互联网的移动医疗系统，其本身就是一种创新。

(2) 移动医疗的架构创新需求

移动医疗系统通过现代移动互联网技术，把原本只能在医院获得的服务延伸到了医院之外。通过移动医疗系统，用户可以方便的享受系统背后的各家医院提供的医疗服务，极大节省在医院所会面临的各种时间浪费。

移动医疗系统本质上就是要打破传统单一医院的约束，实现平台化管理。但是，我们也需要考虑到对于医疗行业而言，所有涉及医生、病人、科室等的业务数据都是作为医院的核心资产保存在医院内部，而且每家医院的业务数据存储和获取方式并不完全相同，这就为我们开展业务带来了挑战，也是技术创新所要面对和解决的问题。

移动医疗系统作为一个统一的大平台，需要完成和各个合作医院的深度对接，考虑到医院系统的多样性，我们的平台设计必须足够的灵活，能够方便快速地完成和各个医院的对接。另外，作为互联网应用，随着对接医院数目的增长，越来越多的用户将会使用我们的服务，所以系统的稳定性、可靠性、高性能也必然成为架构的重点。

移动医疗系统的核心业务流程参考图 6-4，我们可以看到处于中间位置的服

务平台一方面要面向客户端系统，另一方面也可能对外提供服务供各种第三方系统调用。而服务平台的背后则是各家医院，显然服务平台存在的意义和价值就在于它能够完成对这些医院的数据整合。

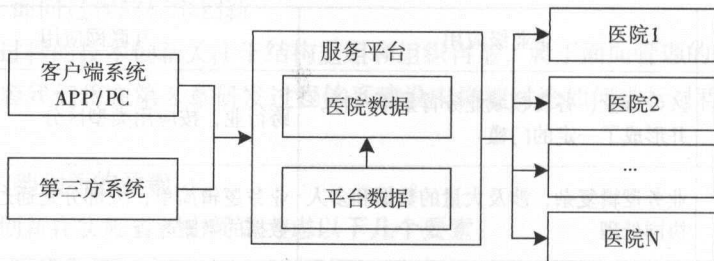


图6-4 移动医疗系统核心业务流程

根据图 6-4，医疗行业固有特点以及互联网环境下的移动医疗系统构建过程中的架构创新需求主要包括：

- 系统集成

系统集成需求主要来自于与服务平台和各家医院之间的数据整合，考虑到各家医院内部可能采用的是不同的 HIS、LIS 或 PACS 系统，势必会面对各种不同的技术实现体系，系统集成的难度就体现在多个异构系统之间的数据传递和转换。

- 服务提供

移动医疗系统中实现服务平台的一大目的在于对外提供灵活、开放的数据服务，这是作为平台的核心价值。这时候，服务平台就相当于位于医院前面的一层服务转换层，对外能够屏蔽各家医院所带有的差异化，并提供统一的数据服务。作为服务提供商，需要考虑第三方系统的入驻，并提供安全的访问入口。

- 系统扩展

对于部分医院，尤其是大型三甲医院而言，一方面它们具备自研能力，在医院内部包装了很多医院甚至科室级别的业务系统，在服务平台的实现过程中需要考虑这些业务系统所带来的特殊性。另一方面，每个医院作为服务平台的一种客户群体，同样也会站在其自身的发展要求上提出很多定制化的需求，这就需要服务平台本身具备较好的系统扩展性。

这些需求都是医疗系统由于互联网化所带来的架构上新的问题，外加互联网应用本身所需要考虑和实现的技术要素，构成了我们在构建新型移动医疗系统的整体创新需求。如果我们能够很好地满足这些需求，就意味着可以通过技术应用达到内部创新。

2. 移动医疗系统架构设计

移动医疗系统架构的设计目标为通过统一的平台来服务所有与我们合作的医

院，并提供尽量标准化的流程。我们从系统的核心组件和整体架构两方面进行切入。

(1) 核心组件

移动医疗系统的核心组件包括：

- 端口适配组件

端口适配组件面向客户端和第三方系统，采用主流的 RESTful 风格构建。即一套后台服务对接多个前端；一种数据结构（如使用 JSON）作为前后端交互的数据媒介；前后端解耦，View 与 Service 高度独立。

- 分布式服务组件

在服务平台内部采用分布式服务构建多系统共用服务，分布式服务的目的就是结合业务流程并充分利用散落在各处的数据。

- 系统集成组件

对于系统集成组件而言，包含两层含义。一方面需要完成与异构系统的对接，并提供异构数据结构的转换功能和统一标准的对外接口服务。另一方面，系统集成表现为一种远程调用，所以提供重要数据本地存储以及系统日志与监控功能。

- 数据管理组件

由于医院对接之后互联网环境下用户体量相较医院内部不是同一个数量级，因此需要封装缓存以提高系统性能。另一方面，通过业务分割实现数据库独立等手段也是数据管理组件需要考虑的内容。

(2) 整体架构与技术关注点

移动医疗系统的创新在于技术创新，即通过使用目前业界主流的面向互联网的架构设计方法和技术体系实现企业级应用到互联网应用的转变，从而实现技术创新。移动医疗系统的整体架构如下 6-5 所示。

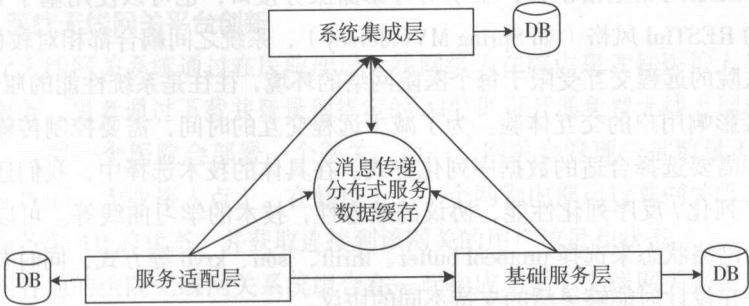


图6-5 移动医疗系统整体架构

在图 6-5 的整体架构中，技术应用上的关注点包括：

- 系统集成交互方式

在系统集成上存在两种集成服务，一种是部署在医院环境的医院集成服务，该集成服务与医院的各种内部系统进行交互；另一个是移动医疗系统自身的平台集成服务，负责从医院集成服务中获取医院数据并做响应处理和回写。

针对医院集成服务和平台集成服务之间的具体交互方式，存在推（Push）模式和拉（Pull）模式两种表现形式。推模式下医院集成服务发现特定事件并通过消息系统推送给平台，而拉模式中平台集成服务主动向医院集成服务发起请求并等待响应。针对目前国内医院的现状，实现过程中一般都是平台集成服务主动去拉取数据的场景居多（见图 6-6）。

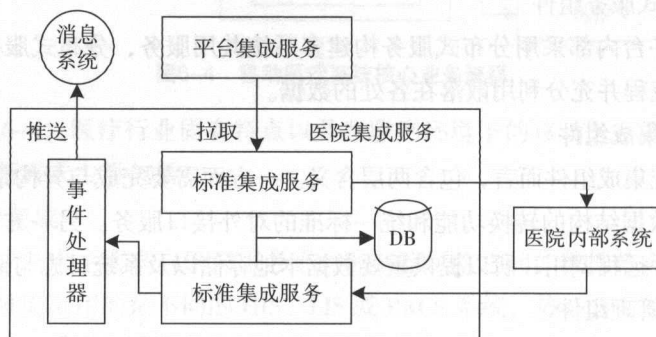


图6-6 两种系统集成服务

· 数据服务实现技术多样性

由于涉及系统集成以及内部系统的各种服务，需要对系统功能进行合理划分，划分的结果就是构成分布式服务环境。在服务与服务之间，可以使用基于 TCP 层 Socket 同步 / 异步调用的 RPC 框架（如 Dubbo 或 Thrift），系统之间紧耦合；也可以使用基于异步消息传递机制的消息中间件（如 ActiveMQ、RabbitMQ 或 Kafka），系统之间完全松耦合；至于对外暴露服务接口，也可以使用基于 HTTP 同步调用的 RESTful 风格（如 Spring MVC/ Jersey），系统之间耦合都相对较松。

与医院的远程交互受限于每个医院网络的环境，往往是系统性能的瓶颈所在，同时直接影响用户的交互体验。为了减少远程交互的时间，需要控制传输数据的大小，即需要选择合适的数据序列化方式。在具体的技术选择中，我们还得考虑对象的序列化 / 反序列化性能，协议的兼容性，技术的学习曲线等。可以根据具体医院的网络状态来选择 protocol buffer、thrift、json、kryo 等方式，同时要求平台集成服务在设计时能够灵活的支持不同的协议。

同时，数据缓存的使用能够提供在互联网环境下数据一致和性能之间的一种平衡性。通过支持多种数据结构的静态数据和动态数据缓存功能，并实现业务级别的自定义缓存就能满足移动医疗系统的数据缓存需求。

· 远程服务可用性监测

由于网络原因或者医院本身提供的对外服务接口的不可用性，将会导致我们提供的某些需要接入医院网络的服务不可用。当这样的情况发生时，为了提高用户的体验，同时避免不必要的麻烦（如付费成功以后才发现医院的挂号服务不可用，而引发的一系列后续烦琐的步骤），移动医疗系统需要及时发现哪些医院的服务不可用，同时能明白的告知用户发生了什么。为此需要提供针对医院服务级别的监控机制。实现服务监控最常见的方法就是心跳检测，图 6-7 展示了一种心跳检测的示意图。

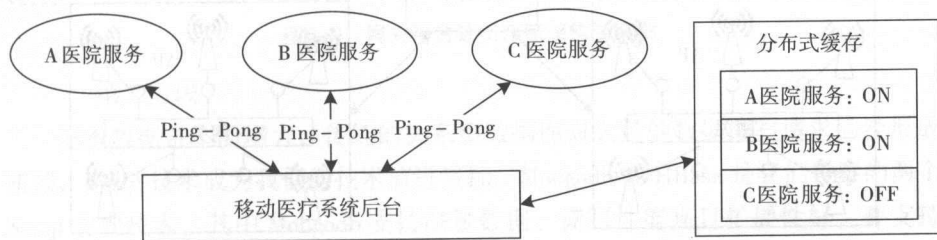


图6-7 心跳检测

移动医疗系统后台通过 ping-pong 机制来检查医院服务的可用性，具体的检查时间可根据具体情况来设置。同时把 ping-pong 检测的结果放入分布式缓存中，这样当平台上提供的某些服务需要和医院交互时，直接通过检查缓存就可以发现服务是否可用，提高了实时响应性。

6.2.3 技术演变创新案例

在本节中我们将通过一个技术演变创新的案例来介绍如何进行内部创新，该案例同样也是属于自主创新的一种表现形式。

1. 医疗无线网关平台创新

医疗无线网关系统通过在医院架设无线网络为在院内患者和医护人员提供无线上网服务，患者通过下载并登录到指定的 APP 即可开通免费无线上网权限。在硬件上，一般一个医院会部署一个网关，而一个网关会管理一批数量不定的 AP（Access Point，无线接入点）。在软件上，一个网管内部运行着网关管理系统，实时监控各个 AP 的状态，并获取连接到该网关的用户数量和状态。

全国各地的医院无线网关系统组合在一起构成了整个无线网关平台。目前全国的医院数以千计，假设该网管平台连接着 500 家医院，而每个医院平均拥有 100 个 AP，那么网关平台将监控 50000 个 AP 的健康状态。AP 实时状态的监控通常采用心跳机制，也就意味着网关平台需要与这些分散在全国各地的网关系

统建立长连接,从而获取其实时健康状态。整个无线网关平台的运行结构如图 6-8 所示。

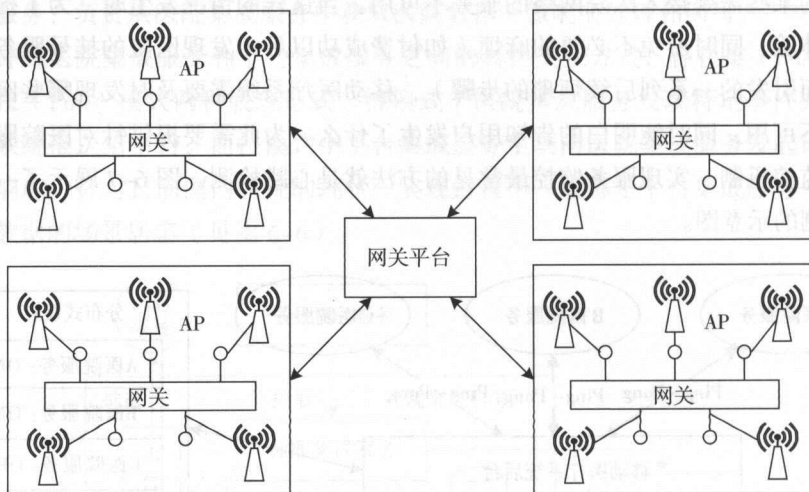


图6-8 无线网关平台运行结构图

另一方面,又假设每一个医院访问 AP 的在线用户维持在几百的数量级,这些用户登录网关进行免费上网、上网时的流量情况以及何时停止上网等信息也需要上报到网关,网关会把这些数据简单处理之后再次上报到网关平台,而网关平台针对这些上报的面向业务和用户体验的数据需要做数据统计分析以便更好地支持产品的优化和营销工作的展开。

从上面的分析我们可以看到,医疗无线网关平台面对的是高并发、大数据量的业务需求,但网关平台所接入的网关数量并不是与生俱来,而是随着业务的演进、市场营销的推动慢慢增加,随之而来整个医疗无线网关平台的技术架构也表现为一个不断演变的过程。

2. 医疗无线网关平台架构演变

医疗无线网关平台架构经历了三种主要的演变,包括数据存储架构演变、服务拆分演变和可用性演变。

(1) 数据存储架构演变

医疗无线网关平台的数据存储架构经历了如图 6-9 中所示的演变过程:

· 第一阶段

第一阶段的存储方案比较简单,就是使用传统的关系型数据库 Mysql。该阶段的平台接入的网关数量不大,相应的数据量也不大。数据在存储的同时也需要支持业务查询。

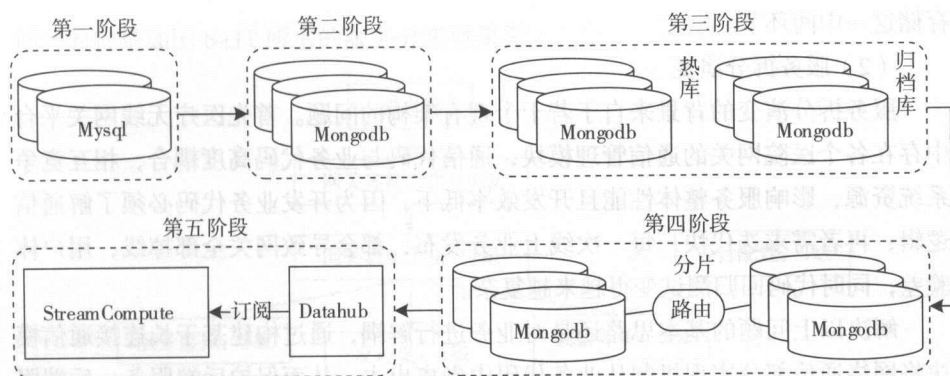


图6-9 网关平台数据存储架构演变图

· 第二阶段

随着数据量不断增大，传统的关系型数据库对大量的数据读写请求已经形成瓶颈，Nosql 技术成为我们的技术演进目标。MongoDB 和 HBase 是我们考察的两个 Nosql 实现技术，其中 MongoDB 支持海量数据、读写性能高且扩展性好，并支持丰富的查询语言。HBase 相较 MongoDB 而言，其对复杂查询的支持较弱，因此这一阶段，Mysql 转变成了 MongoDB，通过 MongoDB 的副本集（Replica Set）功能提供数据存储和查询服务。

· 第三阶段

当数据量进一步增大，通过 MongoDB 的副本集进行全量数据存储和查询也出现了瓶颈，这时候我们的思路是实现数据冷热库分离。所有数据将同时写入到冷热库中，但热库中只存放最近 30 天的数据便于查询统计，而冷库则不进行任何读取操作。因为对于 AP 和用户上网的监控数据具有较强的实效性，冷热库显著提升了数据的查询效率。

· 第四阶段

为了充分利用 MongoDB 的分片（Sharding）功能，确保其横向扩展性，我们将副本集架构调整为分片架构，通过使用更多的磁盘空间用于存储快速增长的数据。这个阶段的数据量已经达到每天单表上千万的量级。

· 第五阶段

当每天的上报数据单表达达到千万级，即使使用分片还是会偶尔造成读写过程中的性能损耗和出错情况。由于存放在 MongoDB 中的数据主要目的并不是用于业务操作，而是面向海量的离线数据分析，因此可以引入一些专门用于数据分析和处理的管道系统实现数据在不需要存储的情况下完成分析工作。离线海量大数据的分析工具也很多，可以选择阿里云的大数据总线 Datahub 来实现这一目的。通过将网关上报数据直接写入 Datahub，并对其数据流进行分析即可省略掉 MongoDB

存储这一中间环节。

(2) 服务拆分演变

服务拆分演变的背景来自于若干个现有架构的问题。首先医疗无线网关平台中存在各个医院网关的通信管理模块，通信代码与业务代码高度耦合，相互竞争系统资源，影响服务整体性能且开发效率低下，因为开发业务代码必须了解通信逻辑；再者需求迭代快，每一次线上业务发布，都会导致网关全部掉线，用户体验差；同时代码回归测试变得越来越复杂。

解决以上问题的基本思路还是对业务进行解耦，通过构建基于长连接通信模块将网络通信部分实现机制从业务代码中剥离出去，从而保护后端服务，后端服务只需实现业务功能即可。通过服务拆分，业务功能的开发和部署就变得轻量级，由于需求变更导致的版本迭代也只需对业务功能进行不断升级即可，通信模块可以基本保持不变。拆分后的通信模块结构如图 6-10 所示，我们可以看到通信模块为后端的各个业务服务提供了网络通信服务并保持自身的高度独立性。

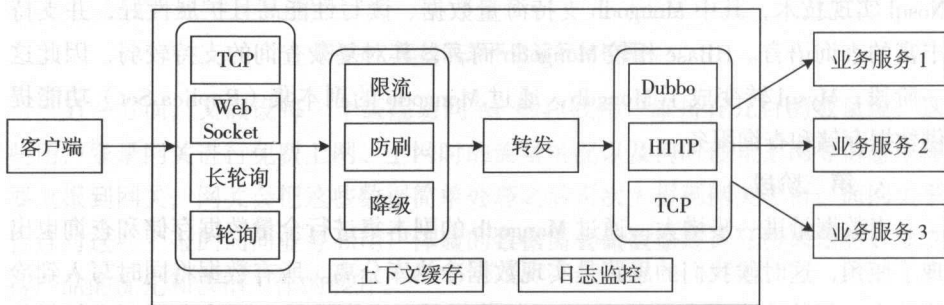


图6-10 网关通信模块结构图

(3) 可用性演变

作为基础性服务，无线网关平台应具备较高的可用性。在可用性上，系统也经历了隔离、服务分级等几次比较大型的演变过程。

在隔离技术上，无线网关平台综合应用线程隔离、进程隔离、机柜隔离和读写隔离等方式，确保服务的稳定性。线程隔离确保当一种业务的请求处理发生问题时，不会将故障扩散到其他线程池，从而保证其他业务可用；进程隔离上将通信模块与各个业务系统进程隔离，避免相互影响；机柜隔离上将集群内的服务部署在多个不同的机柜上；而读写隔离则主要针对 MongoDB 的读写服务隔离。

在服务分级上，将系统服务分成三个不同的等级，一级系统具备完善的容错降级机制及对低级别服务的熔断措施、定期压测、配置高级别的监控告警流程；二级系统多采用异步方式进行系统交互，容忍暂时数据不一致性；三级系统则可随时降级整个服务。基于服务分级思想，对无线网关相关业务进行重新划分之后

得到的就是如图 6-11 所示的服务分级效果图。

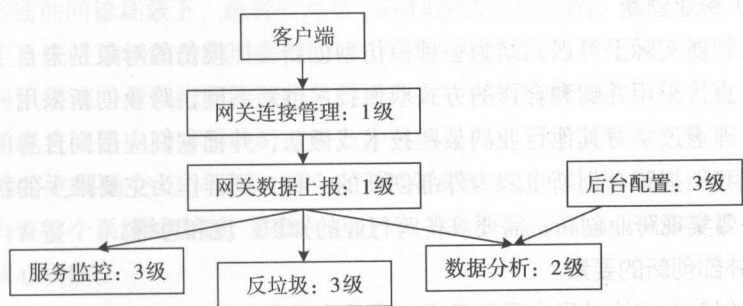


图6-11 服务分级示意图

以上三级服务的拆分来自于对用户的影响，如一级服务会影响到用户对网关的直接使用，二级服务不影响用户上网，但可能会丢失重要数据，并影响体验，三级服务所对应系统的稳定性则对线上业务无影响。

6.3 外部创新

内部创新往往需要较长时间，而且能否在计划时间内通过现有的资源开发出新的产品存在很大的风险。相较内部创新，外部创新能够快速获取创新结果，同时可以将风险转嫁到外部团队。除此之外，外部创新也有其特定的应用场景。如果一个组织的产品已经远远落后于市场，或者其自身对产品的把握没有竞争对手那样有效果，那么通过外部创新的手段提高自身产品的竞争力不失为一种好的选择。

6.3.1 技术外部创新的类型和要素

与内部创新一样，外部创新同样也有类型以及相应的要素。

1. 外部创新的类型

外部创新的类型主要就两种，分别是外部获取和跨业创新。

(1) 外部获取

一家强大的公司可能会选择并购等方式来改变现有的技术体系，但更多的手段是通过与其他公司或组织进行合作以获取新的技术。并购和合作构成了技术外部获取的两种基本方式。如果一家公司没有时间和能力，或者没有建立内部创新的机制，外部获取可能是最直接、也是最快能出成果的创新方式。

并购方式通常关系到公司发展的战略，属于重量级的技术创新手段，技术管理者一般不能主导。而技术合作则相对轻量级，技术管理者通过对市场行业、技

术体系的分析往往就能决定合作对象并推动合作事宜。

（2）跨业创新

跨业创新实际上可以归结为一种模仿型创新，但模仿的对象是来自于外部的行业。与直接采用并购和合作的方式获取技术创新不同，跨业创新采用一种间接的方式，即通过学习其他行业的某些技术或做法，并把它们应用到自身的行业和产品中。我们把跨业创新也归为外部创新的一种，同样作为主要推手的技术管理者，如果要实现跨业创新，需要具备跨行业的知识广度和思维。

2. 外部创新的要素

外部创新在实施过程中需要考虑以下几个要素：

（1）成本

外部创新的成本不仅仅指的是金钱支出，更多的时候是时间和金钱之间的综合考虑。选择外部创新的主要目的就是系统节省创新时间，快速推出产品从而获取技术创新所带来的收益。对于通过合作方式从外部获取技术创新而言，成本是首先需要考虑的问题。有些外部技术服务动辄一年几百万，有些虽然价格还好，但实施时间可能偏长。

（2）风险

只要是与外部交互和对接相关的事宜都要充分考虑风险，因为存在不可控因素。在从外部引入技术的过程中，如果缺乏对本身能力的认识，盲目进行技术创新，会导致失去对核心能力的控制力。另一方面，在与外部企业进行合作过程中，对技术获取过程的控制非常重要，由于没有产权关系，这种控制就显得十分复杂，需要有强大的项目管理和过程协调能力。

（3）现有设施整合

同内部创新一样，外部创新同样涉及现有设施的充分利用，这种利用体现在整合过程上。开发系统性的方法来分享从外部获取的创新技术是整合的一个重要方面，需要确保知识和信息存在各个层次并且应该能够被使用的人所同享。另一方面，在团队组织和文化上，如果新的技术体系对目前的设施有较大冲击，则可能需要进行组织架构的调整，并通过现有的技术体系与外部获取的技术体系之间的碰撞，从中调整和创建新的团队协作文化，对技术本身也可能形成新的知识产权。

6.3.2 技术外部获取案例

在本节中我们将通过一个技术外部获取的案例来介绍如何进行外部创新，该案例属于合作创新的一种表现形式。

1. 智能医生系统简介

在普通的问诊场景下，患者针对某个病理症状提出问题，然后医生进行解答。智能医生系统将该场景从线下搬到了线上，即通过人工智能化的回答方式满足患者的轻问诊需求。可以把智能医生系统看作是一种智能机器人系统，患者与智能机器人系统之间是一对多的映射关系，并采用经典的请求应答方式完成患者问题与医生回答之间的交互。图 6-12 展示了智能医生系统的基本架构，智能医生机器人充当着整个系统的大脑，主要包括分词（Word Segmentation）和搜索（Search）这两个核心功能。

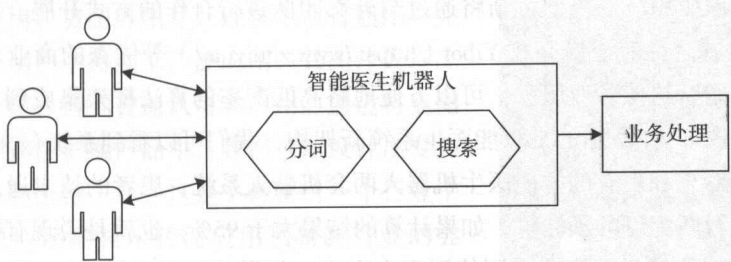


图6-12 智能医生系统基本架构

分词算法上，需要加载词典，然后获取待切分文本（也就是患者输入的问题）并将文本分解为短语。同时，可以使用诸如 Elastic Search 等搜索引擎实现基本的搜索算法。在搜索之前，根据业务需要通常会对一些禁用词、过滤词、同义词和业务词进行预处理。

基于搜索引擎所提供的 TermQuery 等基于评分的搜索功能，并结合业务上的各种关键词匹配，我们设计如图 6-13 所示的搜索框架图，智能医生系统通过该算法框架为患者提供智能问诊服务。

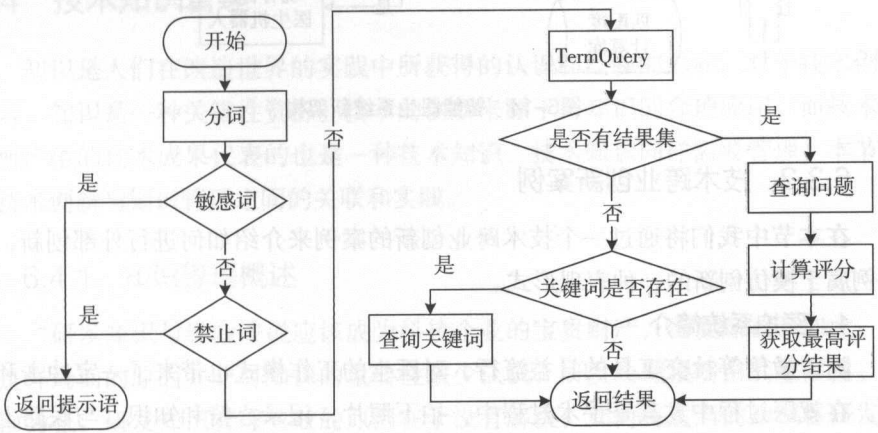


图6-13 智能医生搜索框架图

2. 智能医生系统架构演变

图 6-13 所示的智能医生搜索框架结构，实现成本也不高，但存在一个明显的问题就是匹配率不高。通过搜索引擎自带的搜索功能通常只能提供 60%~70% 的问答匹配率，这在用户体验上并不能提供令人满意的结果，尤其对于问诊场景而言，答案的正确与否会影响到患者的判断，原则上应该提供 95% 以上的匹配率才能基本满足业务场景的需求。但通过技术的调查和预演，为了实现 95% 以上的匹配率，需要对搜索算法做深入研究并实现机器学习基础设施，这已经超出智能医院系统开发团队的技术能力范围，为此我们通过技术外部获取的方式实现技术创新。

智能医生系统的外部创新将通过与外部团队进行合作的方式开展。对于智能机器人领域，目前业界存在 Yibot (<https://www.zhuiyi.ai/>) 等优秀的商业软件，通过购买并整合这些商业服务，可以方便地将高匹配率的算法框架集成到现有的架构中。如图 6-14 所示的是智能医生系统新架构，我们可以看到系统存在 Yibot 智能医生机器人和现有的智能医生机器人两套机器人系统，患者的请求通过前置路由系统进行匹配度的预计算，如果计算的结果大于 95%，也就是说现有智能医生机器人系统能满足搜索需求则使用现有实现，如果不能则调用 Yibot 智能医生机器人服务。这样设计的原因是 Yibot 智能医生机器人服务是根据服务的调用次数进行收费，使用预计算机制可以降低成本。这也是使用外部采购的技术创新方式所需要考虑的一个重要因素。

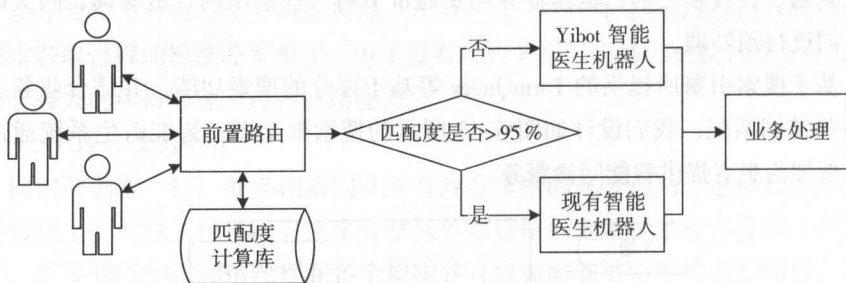


图6-14 智能医生系统新架构

6.3.2 技术跨业创新案例

在本节中我们将通过一个技术跨业创新的案例来介绍如何进行外部创新，该案例属于模仿创新的一种表现形式。

1. 医拍系统简介

随着微信等社交工具的日益流行，对医生的工作模式也带来了一定冲击和变化。在就医过程中尤其是手术过程中，拍下照片，记录心情和知识，与医路上的伙伴们一起分享成为医生工作的一种新方式。一方面，在照片上划动指尖，语音

留言,有些精彩是文字无法表达、也是传统院内软件所无法提供的功能。另一方面,在手术场景下,由于时间紧、影响大,医生对于拍照并传播的速度、易用性等用户体验会有更高的要求。再则,关注话题和朋友,通过微信进行信息的分享并从中获取更多的认可和知名度也是医生的一种主观愿望。在这种背景下,医拍系统就有了其特定的应用场景和需求。

2. 拍标工具产品设计

医拍工具的产品设计和规划上,可以参考美图(<http://www.meitu.com/>)等图片处理技术在拍标上的应用。美图等图片处理技术在普通用户尤其是女士用户中得到了广泛应用,其图片处理工具和操作模式得到普遍认可和应用。通过将美图技术应用到医护端产品中,我们可以设计出专为医生打造的医拍工具,这是一种将另一个行业中已经成熟应用的技术转而应用到崭新行业的尝试,属于技术跨界创新。图 6-15 展示了使用图片处理技术所实现的医拍工具界面。

从图 6-15 中,我们不难看出那些熟悉的用于画图、语音、文字、马赛克等操作按钮和方式,这些操作按钮和方式几乎不需要任何学习成本,并能帮助医生在手术等紧张环境下快速拍摄图片、加工处理并分享到朋友圈。

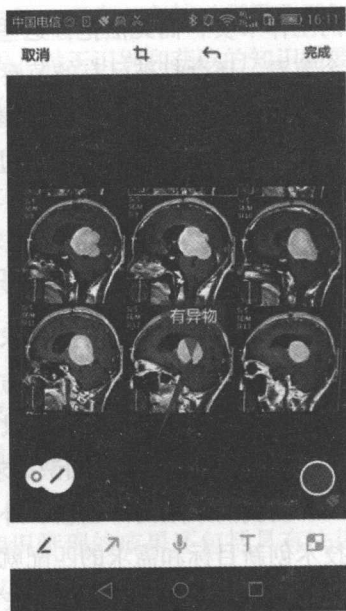


图6-15 医拍界面

6.4 技术知识管理

知识是人们在改造世界的实践中所获得的认识和经验的总和。对于技术创新而言,知识是一种关键性资源。技术的创新来源于对知识的合理应用,而技术创新所产生的技术成果代表的也是一种技术知识。技术知识同样需要管理,本节讨论技术创新与知识管理之间的关联和实践。

6.4.1 知识管理概述

“研发知识与经验按说应该成为科技企业的宝贵财产,而实际并非如此,造成这样局面的原因并非对知识的重要性缺少认识,研发经理很明白其重要性。到目前为止,研发知识管理失败的原因在于没有解决:如何将适当的知识在研发人员需要此知识时提供给他。”这是国际知识管理领域的权威 Michael E. McGrath (集

成开发模型 IPD 的前身 PACE 理论的创立者）说过的一句话，代表着研发知识管理所面临的困惑，这种困惑在国内中小型企业尤其是初创型企业表现得尤为明显。

要解决这个困惑，我们首先要明确知识管理要做什么？知识管理把“隐性知识显性化”，是知识库、交流、环境之间的综合体，所管理的知识将作为一个团队组织中过程资产的重要组成部分。对于软件研发而言，所谓隐性知识，通常指的就是在那些业务人员和技术人员脑中的蓝图，把握这些蓝图的人就能形成一定的工作节奏，而无法把握这些蓝图的人如果加入到团队中，不但可能适应不了这个节奏，还会打破已有的节奏，这就是我们要进行研发知识管理的目的所在。

技术创新过程中伴随着很多技术知识的产生，这些技术知识很多具有创新性和实用性，相较普通的研发过程知识，这些创新性的技术知识更加需要保存、转移和转化。

6.4.2 技术创新与知识管理

成功的技术创新管理要求企业能够有效管理从学习中所获取的知识，这实际上体现的是一种需求。另一方面，知识管理同样需要过程并形成一定的成果，这些成果代表着技术创新和知识管理的结合体。

1. 技术创新中的知识需求

当我们采用内部创新、外部创新或两者结合的创新模式时，知识管理方法与技术创新目标和需求的匹配就显得很重要。无论是进行根本性创新还是渐进式的改进，对于一个团队和组织来说，知识管理都是需要考虑的一个重要方面。

图 6-16 描述了技术创新过程中，开发和实施知识管理需要考虑的需求，这些需求按照内部创新、外部创新这两个维度，以及知识本身的复杂度进行排列组合，形成了知识管理需求的四象限模型。

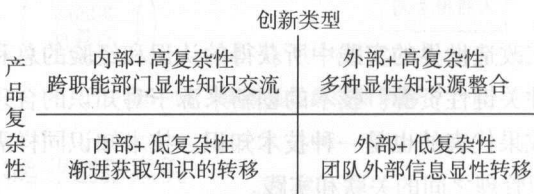


图6-16 知识管理需求

图 6-16 中的横轴代表技术创新的类型，对于外部创新而言，知识的获取和管理存在较大不确定性，隐性知识到显性知识的转变和转移显得尤为重要；而对于内部创新而言，知识管理更多在于实现良好的跨部门协作。

另一方面，图 6-16 中的纵轴代表知识的复杂性，复杂性越高，知识管理方法就越是重要，需要增加更多的沟通以及更多层级的相互传递。显然，复杂性往

往也需要增加更多跨职能部门之间的有效协作。

2. 知识管理的过程

如同项目管理中的范围、时间和成本管理，知识管理也应该落实到每一位研发人员，表现为一系列的过程实施和监控。针对技术创新，知识管理涉及以下三大过程：

（1）存储

知识存储过程的主要目标在于知识源的识别和获取。通常，存储过程需要开展跨职能部门之间的交流以获取全面且有效的系统知识。对于识别和获取的知识需要一个空间或平台进行统一管理，带有版本控制功能的系统是知识存储的必备工具。

（2）搜索

知识搜索是一个知识扩展的过程，通过选择性的识别所需要使用的知识，进而对这些知识进行评估，可以为技术创新提供决策支持，进而形成一致和完整的技术评审技术。

（3）再使用

知识再使用的过程是一个监控和调整的过程，当面对一种新的技术创新需求时，充分利用原有的知识积累，在与现有知识需求的合并中利用知识是其主要目标。换言之知识再使用是管理知识变化的过程，通过增强研发人员之间的沟通促进团队的知识沉淀。

除了团队中不断累积的知识存储，技术创新知识管理的成果还包括具有法律效力的软件著作权与专利的申请，关于软件著作权与专利的讨论超出了本书的范围，读者可按照需求了解相关的法规和流程。

6.5 本章小结

技术创新与产品演进密切相关，作为技术管理者应该了解和掌握技术创新的常见类型并应用到具体的产品开发过程中。本章从技术变革的基本规律出发，讨论技术创新的策略和模式，并把技术创新工作归为内部创新和外部创新两大类。

内部创新包括面向基础研究的技术创新、面向产品的技术创新和面向过程的技术创新等不同的表现形式，本章围绕移动医疗系统的开发历程，以技术应用和技术演变为切入点内部创新的实现方式做了介绍并提供相关案例。对于外部创新，同样存在不同的类型，我们也对技术的外部获取和技术的跨业创新提供了案例分析。

另一方面，伴随着技术创新会产生一系列的技术知识，本章最后也对技术创新中的知识需求以及知识管理的过程做了介绍。

管理体系篇

向技术管理者转型

软件开发人员跨越行业、技术、管理的转型思维与实践

本篇共有三章，介绍作为一名技术管理者所应掌握的各项管理技能，包括：

1. 软件项目管理。产品通过一系列的项目得以实现和发展。对于项目管理而言、范围、时间管理可以拆解成更加面向研发团队的需求和计划管理，同时在介绍与研发过程密切相关的质量管理和风险管理基础之上，专门针对交付管理这一与技术管理者日常工作密切相关的主题做单独的分析 and 介绍。

2. 研发过程体系建设。研发过程体系建设是作为技术管理者开展技术管理工作最重要的环节。研发过程体系建设在思路先梳理典型的软件过程模型，然后对目前软件开发领域最常见的敏捷方法进行详细展开，再采用过程改进思想来指导日常工作管理过程。

3. 组织管理。相较过程管理，组织管理更加偏向与人而不是事。面对大部分组织场景，可以将技术管理者眼中的组织管理划分成向下管理、向上管理、向外管理和自我管理四个主要方面。

管理是技术管理者手上最重要的武器，以上三章内容分别构成了技术管理者在管理手段上的各项“硬”技能和“软”技能。

7 软件项目管理

从系统工程角度，软件开发可以分成三大部分，即软件工程的组成三段论（见图 7-1）。软件实现主要回答“客户需求如何通过软件系统来表达”这一问题，包括需求工程和软件设计两大部分；项目管理则从范围、时间、成本等角度出发讨论如何在一定的约束条件下实现系统，包括计划和估算管理、质量与配置管理、风险与团队管理等内容；而过程改进则围绕软件开发的过程，提出持续优化的方法和实践确保得到令人满意的结果。

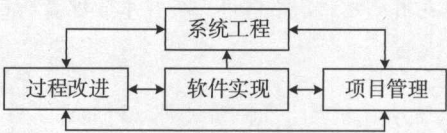


图7-1 软件工程组成三段论

软件工程围绕软件生命周期进行展开，经典的软件生命周期包括可行性研究、需求分析、概要设计、详细设计、编码、测试、发布、维护等各个环节。通常，我们以详细设计为界限把软件工程的环节分成上游工程和下游工程两大部分。同时，围绕如何组织这些环节也有一批软件生命周期模型，例如瀑布模型、敏捷模型等代表性表现形式。

本章围绕系统工程中的项目管理维度展开讨论，关于过程管理与改进我们将在下一章研发过程体系建设中进行介绍，而关于软件实现的大部分内容我们已经在本书第 4 章和第 5 章中做了阐述。

7.1 项目管理体系概述

项目管理包含一整套知识体系，如业界具有代表性的 PMBOK（Project Management Body Of Knowledge，项目管理知识体系）就是对项目管理所需的知识、技能和工具进行的概括性描述^[9]。用最简单的方式来概括 PMBOK 中的内容就是五大过程组 + 十大知识领域。其中五大过程组代表着一个项目从开始到结束的生命

周期中应该经历各个阶段（见图 7-2）。

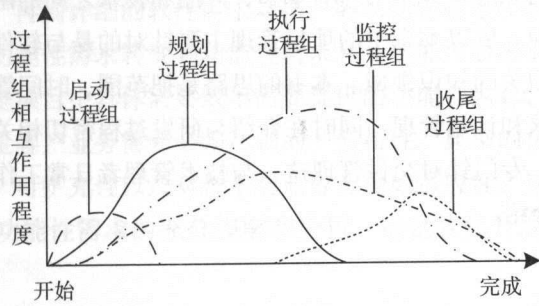


图7-2 项目管理五大过程组以及相互关系

按照 PMBOK 第五版中的表述，项目管理十大知识领域分别是：项目整合管理，其作用是保证各种项目要素协调运作，对冲突目标进行权衡折中，最大限度满足项目相关人员的利益要求和期望；范围管理，其作用是保证项目计划包括且仅包括为成功完成项目所需要进行的所有工作；时间管理，其作用是保证在规定时间内完成项目；成本管理，其作用是保证在规定预算内完成项目；质量管理，其作用是保证满足承诺的项目质量要求；人力资源管理，其作用是保证最有效地使用项目人力资源完成项目活动；沟通管理，其作用是保证及时准确地产生、收集、传播以及最终处理项目信息；风险管理，其作用是识别、分析以及对项目风险做出响应；采购管理，其作用是从机构外获得项目所需的产品和服务；干系人管理，其作用是识别干系人、规划干系人管理策略并使用该策略管理和控制干系人的参与。

以上这十大知识领域之间的关系可以见图 7-3。

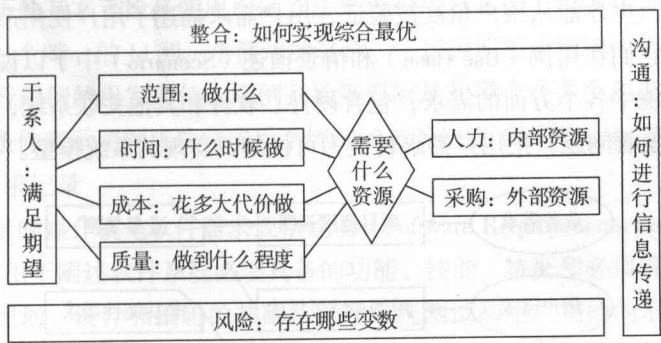


图7-3 项目管理十大知识领域之间的关系

PMBOK 面向通用的、全行业的项目管理过程，对于软件开发而言，由于一般不涉及材料的消耗，不需要对成本和采购做专门的管理；人力资源、沟通管理

和干系人管理涉及技术管理者的组织管理和软能力，我们会在第9章做这方面的阐述；整合管理的思想贯穿本书所有章节，产品和技术之间的融合、系统的集成等都是其具体体现。所以本章中的项目管理主要针对的是与软件开发和技术管理者日常工作密切相关的知识领域。本书的思路是把范围、时间管理拆解成更加面向研发团队的需求和计划管理，同时在介绍与研发过程密切相关的质量管理和风险管理基础之上，专门针对交付管理这一与技术管理者日常工作密切相关的主题做单独的分析 and 介绍。

7.2 需求管理

需求是对软件的描述，规定软件的功能及其运行环境。在软件行业，由于不充分的需求规范、需求的随意改变、缺乏需求管理等因素是影响软件开发成败的关键之一，因为对于软件开发而言，需求是源头。这就意味着我们需要对需求有足够的重视，需求本身也是一项工程。

7.2.1 需求工程

1. 需求层次

如同第5章介绍的软件架构，我们对需求进行分析之后会发现其同样具备一定的层次性，层次性的产生原因一方面在于需求抽象的过程，另一方面也取决于需求传递过程中所流转的角色和媒介。从图7-4中可以看出，业务需求位于整个需求层次的最上层，代表的是客观存在的业务实体本身，反映了组织机构或客户对系统、产品高层次的业务利益和目标要求。但业务实体只有被抽象成用户需求才能被识别，也才能从用户角度被验证，用户需求描述了用户使用产品必须要完成的任务，它们在用例（Use Case）和情景描述（Scenario）中予以说明。系统需求描述了系统中各个方面的需求，包含硬件、软件和其他关联系统，更多站在技术人员

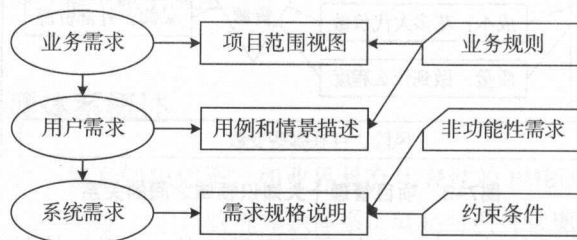


图7-4 需求的层次

需求的不同层次分别会有不同的需求表现形式，从最初的项目视图和功能范围，到功能用例，再到详细的软件需求规格，需求的粒度逐步细化，而对功能的描述也从纯粹的功能性需求转变到功能性需求和非功能性需求的结合体。所谓功能性需求，指的是项目中具体需要或不需要的功能和内容，而非功能需求则代表项目中为满足客户业务需要必须达到的一些特性，典型的包括系统性能、可靠性、可维护性、可扩充性以及技术与业务方面的适应性。这些非功能性需求的提炼依赖于对功能性需求的充分理解和分析，也是软件开发人员的一项关键职责。

2. 需求开发

需求是一项工程，同样是一个开发的过程，从获取需求到需求的最终验证，需求开发构成了一个闭环框架（见图 7-5）。

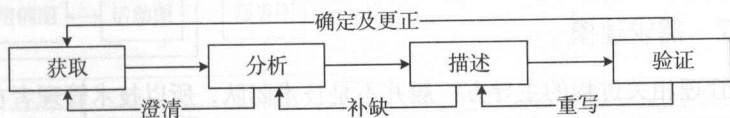


图7-5 需求开发过程框架

（1）需求获取

需求获取的手段包括访谈、工作坊（Workshop）、焦点小组（Focus Group）、观察、问卷调查、系统接口分析和用户界面分析等。技术管理者通常是这一过程的参与者而不是主导者，但为了更好地理解需求，技术管理者有时候也会以一定正式或非正式的形式组织需求获取活动。

（2）需求分析

需求分析的基本手段是使用用例方法来理解用户需求。用例在表示上应该有统一规范，一个唯一的 Id 和一个简洁的名称、一段说明表示用例的意图的简短文字、开始执行用例的出发条件、用例开始需要满足的零个或多个前提条件、一个或多个后置条件表示用例成功完成后系统的状态等构成了用例的基本要素。

（3）需求记录

需求记录的媒介就是软件需求规格说明（Software Requirements Specification, SRS），SRS 用于阐述软件系统必须具备的功能、性能、特征和必须遵循的约束，是后续项目规划、设计和编码的基础，也是系统测试和用户文档的基础。SRS 一般建议包含用户界面设计但不应该包含设计、构建、测试或项目管理方面的细节。

（4）需求确认

需求确认的基本思想是评审（Review），通过评审会议对每一个需求利用检查表（Checklist）思想进行确认。

通过以上流程获取的需求可能还会面临很多问题，典型的包括不适当的需求、无足够用户参与、忽略用户分类、模棱两可的需求说明、不必要的特性（镀金）、过分精简的规格说明等。这些问题中部分可以在开发阶段进行解决，而剩下的则需要依靠需求管理的理念和方法。

3. 需求管理

需求管理本质在于明确变更控制与边界，即当需求开发完成之后，如何应对业务变更导致的影响。在需求管理中存在一种重要概念，即需求基线（Baseline），我们已经在 3.1.2 节中介绍了需求基线的相关概念。基线中的需求处于配置管理之下，后续的变更只能通过定义的变更控制流程进行，要实现这一目标，需求和代码一样也需要版本控制。关于版本控制和配置管理我们在本章后续内容中还会具体展开。

7.2.2 需求建模

需求管理相关过程的主导方一般并不是技术团队，所以技术管理者在需求管理中往往起到辅助作用。但是需求管理是一个为了更加有序的得到更加结构化需求而不断演进的过程，正如前文所分析的需求层次，业务需求和用户需求的获取和提炼依赖于产品和运营团队，但是系统需求的表现则更多偏向于技术本身，这也是为什么要把需求管理作为技术管理者所应具备的一项技能之一。

技术人员使用模型来表述需求，从需求的层次上，建模的主要对象是系统需求。而模型是一个抽象概念，需要借助于特定工具进行表述，目前使用最广泛的建模工具是 UML。

1. UML

UML（Unified Modeling Language，统一建模语言）为面向对象软件设计提供统一的、标准的、可视化的建模语言。UML 中的事物包括类、接口、用例、组件和节点等结构事物，也包括交互、状态机等行为事物。UML 提供了 9 种图形来表述这些事物，基于事物的结构和行为特征，这些图也可以分成结构和行为两大类，各个图的定义以及分类可参考表 7-1。

表7-1 UML图

名称	描述	小类	大类
类图	类以及类之前的相互关系	静态图	结构
对象图	对象以及对象之前的相互关系		
组件图	组件及其相互依赖关系	实现图	
部署图	组件在各个节点上的部署		

续表

名称	描述	小类	大类
时序图	强调时间顺序的交互图	交互图	行为
协作图	强调对象协作的交互图		
状态图	类所经历的各种状态	行为图	
活动图	工作流程的模型		
用例图	需求捕获和描述	用例图	

软件实现包括上游工程和下游工程，从图 7-6 中的 UML 图之间的关系可以看出，使用 UML 进行系统建模包含上游工程中的需求分析、概要设计和下游工程中的详细设计，意味着从需求到设计我们都可以借助于 UML 来实现系统模型。

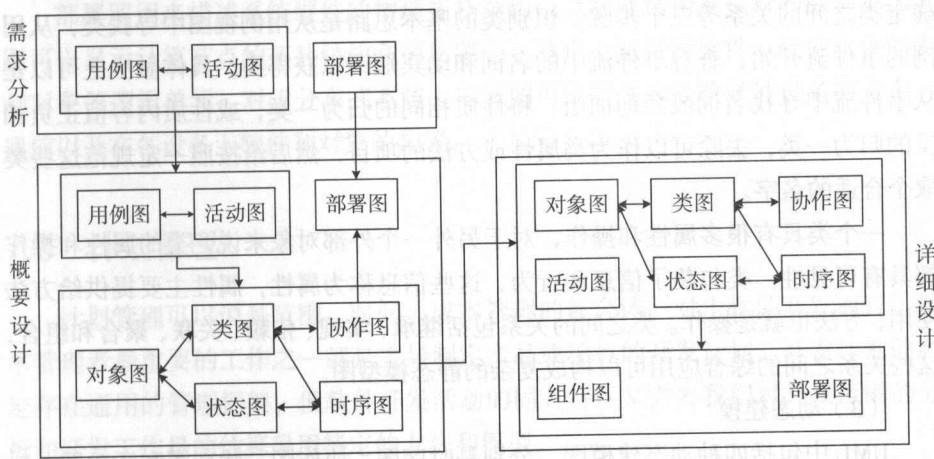


图7-6 UML图关系

2. 系统建模

基于 UML 的系统建模方法一般包括用例建模、静态建模、动态建模和架构建模四大类。

(1) 用例建模

UML 中提倡以用例为中心组织需求，所谓用例，简单讲就是做一件事情。做一件事情需要做一系列的活动，可以有很多不同的方法和步骤，也可能会遇到各种意外情况。因此很多不同情况的集合构成了场景，一个场景就是一个用例的实例。一个用例是用户与软件系统之间的一次典型交互作用，在 UML 中用例被定义成系统执行的一系列动作，也就是系统功能。

系统用例提取之后形成了用例图，但仅仅依靠用例图表现力往往不足，通常都需要在用例图的基础上添加用例描述。用例图中用例名称、用例标识、涉及的角色以及基本描述是必要元素，复杂用例的规格说明则还包含前置条件（Pre

Conditions）、后置条件（Post Conditions）、正常事件流（Flow of Events）、备选事件流（Alternate Flow）以及其他功能需求、设计约束、尚存在的问题等内容^[23]。

（2）静态建模

静态建模的目的是识别系统的静态元素以及它们之间的关系，根据开发系统的静态结构需要创建类图和对象图。类图用需要出现在系统内的不同的类来描述系统的静态结构，包含类和它们之间的关系，类名、类的属性和类的操作是类图的基本组成。对象则对应于真实世界中的某个客观实体，所有的对象都是有唯一标识的独立实体，对象图描述一段时间里特定实例的静态结构。静态建模的主要产出一般是类图。

围绕类和对象展开的静态建模过程可以分为识别类、确定类的属性和操作、确定类之间的关系等三个步骤。识别类的基本思路是从用例视图中寻找类，从用例的事件流开始，查看事件流中的名词和动宾短语以获得类。具体做法上可以是从事件流中寻找名词或名词词组，将性质相同的归为一类，或性质内容值正负相反的归为一类，去除可以作为类属性或方法的项目，然后给按照一定规范这些类取个合适的名字。

一个类具有很多属性和操作，对于另外一个外部对象来说，类的属性和操作都具有可见性。类包装了信息和行为，这些信息称为属性，属性主要提供给方法使用，方法也就是操作。类之间的关系包括继承、实现、依赖、关联、聚合和组合，这些关系之间的综合应用可以构成复杂的静态模型图。

（3）动态建模

UML 中包括四种动态建模图，分别是时序图、协作图、状态图和活动图。时序图表示类之间的交互，这种交互代表了类之间消息交互的顺序；协作图表示类的集合以及这些类发送和接收的消息，即类与类之间的交互协作过程；状态图表示类的行为，在执行动作时它描述类的状态和响应；活动图描述类的活动，它可以用于表示多个类的状态以及它们之间的关系。

时序图和协作图可以合称为交互图，交互图常常用来描述一个用例的行为，显示该用例中所涉及的对象以及这些对象之间的消息传递情况。交互图的基本内容元素包括对象和消息，创建交互图的过程也就是寻找对象、寻找角色和添加消息的过程。在表述上，时序图是一种二维图，纵向是时间轴，横轴则代表了在协作中各独立对象的类元角色，着重体现对象间消息传递的时间顺序，而协作图则强调参与一个交互对象的组织。

状态图描述一个对象在其生命周期中的行为。状态节点通过转移连接的图，描述了一个特定对象的所有可能状态，以及由于各种事件的发生而引起状态之间的转移。当对象的状态数目有限时，就可以用状态图来建模对象的行为，状态图

显示了单个类的生命周期。状态图通常由初始状态、终止状态、活动以及活动之间的转换过程所组成。

活动图也就是通常所说的流程图，使用流程来描述用户事件流，包含状态、分支等要素。活动图中也可以添加泳道（Swimlane），添加了泳道的活动图更能从角色维度区分流程接口的所属关系。

（4）架构建模

组件图很大程度上代表了软件系统实现的不同方面，包含模型代码库、可执行文件、运行库和其他组件的信息。组件是代码的实际物理模块，UML 中的组件图显示组件以及它们之间的依赖关系，它可以用来表现程序代码如何分解成模块或组件。取决于组件中类的关系，组件之间也有泛化关系和依赖关系之分。

部署图用来描述系统硬件的物理拓扑结构以及在此结构上执行的软件。部署图可以显示计算节点的拓扑结构和通信路径、节点上运行的软件、软件包含的类和对象等逻辑单元。对于分布式系统，部署图可以清楚地描绘硬件设备的分布、通信以及在各设备上软件和对象的配置。

7.3 计划管理

计划管理可以说是范围、时间、成本管理的整合体，对于软件开发而言，技术管理者最重要的工作之一就是主导制定产品或项目的开发计划。开发计划的制定存在通用的管理框架，但软件开发活动的固有特性又需要我们对开发范围的分析 and 开发工作量的估算采用特定的方法和模型。

7.3.1 通用计划管理活动框架

软件项目的计划管理起点在于系统需求，终点在于进度计划。识别开发活动→排列活动顺序→估算活动资源和时间→制定进度计划构成了项目计划的完整开发框架。

（1）识别开发活动

WBS（Work Breakdown Structure，工作分解结构）是面向可交付成果的对项目工作的层次化分解，WBS 有机地组织和定义了项目的整个范围，将项目工作分解（Decomposition）成较小的、更易于管理的多项工作，而每下降一层代表对项目工作更详细的定义。WBS 的各个组成部分有助于系人理解项目可交付成果，许多组织有标准的 WBS 分解模版用于对包括业务需求和管理工作的所有交付物进行分解并得到开发活动。

显然，分解的思想很简单，但分解的层次和粒度却并不容易把控。在 WBS 中

有工作包(Work Package)的概念,代表WBS最底层的可交付成果或项目工作成分。所有的进度安排和监控都是围绕工作包展开。图7-7是WBS与工作包的一个示例,我们可以看到在项目的管理的角度,活动不仅仅指系统的设计和实现,所有围绕最终交付的项目管理、需求等工作都属于需要识别的范围之内。

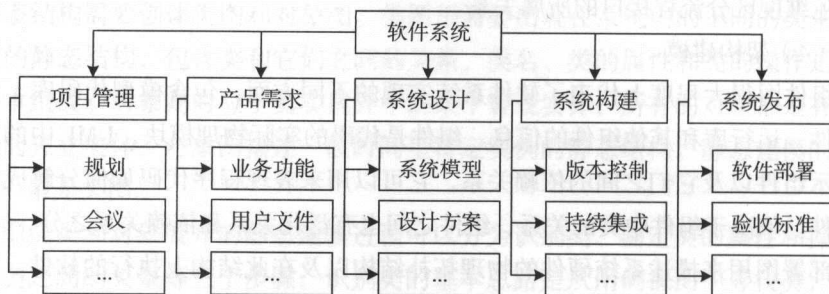


图7-7 WBS与工作包示例

活动通过对工作包的识别与分解得出,是估算、进度制定、执行和监控项目的基础,活动具备活动标志符、前导活动、后续活动、逻辑关系、提前与滞后、资源要求、强加的日期、约束和假设等多个属性,这些属性为后续的活动排序、资源和时间估算提供依据。针对软件开发范围的分解存在一些工程实践,这部分内容我们将在下一节7.3.2中进行详细展开。

(2) 排列活动顺序

活动排序方法常见的包括按照工作的客观规律排序、按照项目目标的要求排序、按照轻重缓急排序、根据项目本身的内在关系来排序。活动之间本身存在完成到开始、开始到开始、完成到完成、开始到完成等依赖关系,也可以在各种关系中添加提前量或滞后量约束,这些都属于强制性依赖关系,一般无法改变。

在具体应用领域可能存在一些最佳惯例,对某些特殊方面,即使存在其他可接受的活动排序方案,也期望采用特定的专门顺序,这些属于非强制性的软逻辑。而类似软件发布与硬件到货这种依赖于其他因素的外部关系通常也是活动排序中的不可控因素,需要特别注意并提前介入。

(3) 估算活动资源和时间

资源估算考虑需要什么资源、什么时候需要、需要多少、获得所需资源由谁拍板等问题。对于软件开发而言,资源主要就是人本身,比较简单。

时间的估算永远都是不准确的,对于软件开发而言,变化所导致的不确定性因素对于估算的影响尤为明显。业界有一些通用的方法可以应用到软件开发的估算中,而软件行业本身也有专门的工作量估算技术,这部分内容我们将在7.3.3中进行详细展开。

(4) 制定进度计划

关键路径法（Critical Path Method, CPM）是制定进度计划中常用的技术方法，图 7-8 是关键路径法的一个示例，通过梳理各个活动之间的依赖关系可以确定完成这些活动的最短时间以及活动之间可以并行操作的程度。通过关键路径法得到的进度计划的最终表现形式通常是甘特图。

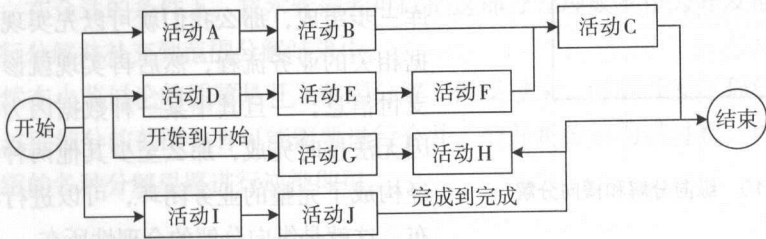


图7-8 关键路径法和甘特图示例

7.3.2 开发范围分解技术

开发范围分解的一大作用是用于工作量估算，因为太大的开发任务通常都比较难以正确估算。另一方面，分解之后的开发范围也为整个开发团队提供一份统一视图，用于监控和管理开发范围，防止出现范围潜变现象。范围潜变是指客户不断提出小的、不易察觉的范围改变，如果不加控制，累计起来导致项目严重偏离既定的范围基准，导致项目失控和失败。对于范围潜变，通过范围的分解和透明化有助于减低其带来的不良影响。

1. 范围分解的误区

根据模块和功能分解开发范围可能是最常见也最直接的分解方法，这种分解方法往往与具体的团队组织架构和开发资源安排有密切关系。按照分解的结果，范围的最底层表现形式是活动（见图 7-9），范围最终由一系列的活动构成，可以把活动指派给具体开发人员。

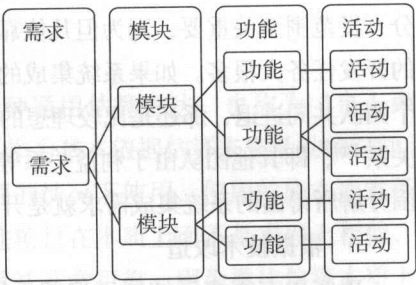


图7-9 根据模块和功能分解

上述分解方法实际上存在一个比较严重的问题，即采用了横向分解方法。在分解范围时，我们应该使用纵向分解而不是横向分解^[24]，所谓横向方式就是根据业务流程把一个大流程分解成几步小流程，这些小流程合起来才能完成整个大流程。如图 7-10 所示，假如我们有个业务流程由获取数据、验证数据和写入数据库三步构成，则这三步就可能被分解成三个任务。试想如果其中有任何一个任务无法

	患者数据	就诊信息	支付信息
获取数据			
验证数据			
写入数据库			

图7-10 纵向分解和横向分解

按时完成也就意味着整个业务流程都无法完整，开发风险就会很大。如果我们换一种方式，把这个业务流程中需要处理的数据分成患者数据、就诊信息和支付信息，而每一种数据都是由获取数据、验证数据和写入数据库三步组成，那么我们就可以先实现患者数据相关的业务流程，然后再实现就诊信息和支付信息，一旦其中某一种数据因为各种原因无法按时完成，那么至少其他两种数据已经构成了完整的业务闭环，可以进行独立发布。这就是纵向分解的合理性所在。

2. 范围分解的方法

上述纵向分解的例子中体现的就是根据数据边界进行范围划分的思想。在敏捷大师 Mike Cohn 的著作《敏捷估计与规划》^[25]中提到了这一分解方法，另外他还认为根据操作边界分解、去除横切（Cross-cutting）考虑、区分功能性与非功能性需求、根据不同优先级进行分解等方法也都是有效的分解方法。考虑到软件开发的固有特性和不同场景，我们认为以下分解方法也有助于更好地明确开发范围，并确保开发人员能够在范围管理中起到一定的推动作用：

· 根据系统集成

很多软件系统都涉及系统集成需求，包括与外部第三方系统的对接，也包括组织内部各个团队之间的功能或数据集成。在对范围分解过程中，识别并分解这部分开发范围至关重要，因为但凡依靠其他团队才能完成的功能其风险性都会比普通的开发任务大很多。如果系统集成的双方是合作关系，即分别处于上游和下游的两个团队共同进退，那还是比较理想的关系。而如果两种之间是遵奉者（Conformist）关系^[26]，即其他团队由于利益关系等因素并不想或没有能力推动系统集成，那么范围分解所得出的系统集成需求就是开发过程中需要重点管理的对象。

· 根据技术改造

现实中有很多系统属于遗留系统（Legacy System），这些系统中普遍存在很多技术债务（Technical Debt），需要通过不断的重构来改善系统架构。但通常，范围的分解面向业务功能，而不是技术改造，如果处于技术债务较多的遗留系统中，技术管理者就应具备从功能开发范围中提炼技术改造需求的能力，并把它们作为开发范围的一部分展示给所有相关人员，以争取开发资源。

· 根据技术试验

有时候，开发人员无法正确地把控开发所需的技术需求。在这种情况下，可

以创建技术预演和试验相关的需求,我们把这种需求称为试验性需求。这种需求通常具备较大的灵活性,因此在范围分解以及后续的计划安排中需要专门考虑。

- 根据管理成本

软件开发范围分解过程中一个比较明显的特点就是几乎不大考虑管理软件开发过程的成本。实际上,按照 WBS 的规范,管理成本也是一种范围,也需要进行分解。在合适的条件下,技术管理者可以把这部分管理成本作为开发范围的一部分进行分解并补充到范围分解结果中。

虽然本小节讨论的话题是开发范围分解,但反过来,随着开发过程的演进,对一些过度细分的范围也有可能需要进行合并。合并是分解的逆过程,可以按照前面介绍的各种分解思路进行逆推即可。

7.3.3 开发工作量估算技术

开发工作量估算在软件开发领域是一个老大难问题,因为软件开发是一项需要创新的工作,并不能像传统制造业一样容易对工作进行量化管理。一方面软件开发过程中新技术的不断出现和应用,软件本质是复杂和不可见的。另一方面,传统项目进程可以用相近的项目做参考,但软件项目在绝大多数情况下都是独一无二的,缺少可以参考的经验数据。

但软件估算仍然存在一些典型的估算策略,包括常见的基于分解的自顶向下和自底而上策略,类比策略、代理策略等。基于这些策略,业界诞生了很多估算技术,我们把这些估算技术分成普通的通用估算技术和针对软件开发的专用估算技术。

1. 通用估算技术

(1) 类比估算法

类比估算法(Analogous Estimating)是一种通用估算方法,也称为“自上而下的估算”,是指把以往类似活动的实际时间作为基本依据估算未来活动的时间。类比估算经常在项目早期等项目详细信息有限的情况下使用,使用的成本通常低于其他方法,但精度也较低。有些项目与以往项目在本质上而不是表面上相似,估算者如果掌握必要的专门技术,则类比估算法非常可靠,因为类比估算本质上属于专家判断的一种形式。

(2) 参数估算法

参数估算法(Parametric Estimating)也是一种通用估算方法,应用了生产率概念。所谓生产率是指生产单位成果在单位资源下所需要花费的时间,而活动时间 = 活动数量 * 生产率 / 可用资源数量。参数估算法相较类比估算法能提供更为可靠的估算结果,但依赖的条件也更多,在建模的历史资料准确、模型中的参数

容易量化、模型具有对大小项目都适用的可缩放性时比较可靠，其准确性取决于模型的复杂性以及作为模型一部分的资源数量和成本数据。

(3) 专家判断法

专家判断就是指具有应用领域或者开发环境知识的人员对工作进行评估。通过借鉴历史数据，专家判断可以提供估算所需的信息或根据以往类似项目的经验，给出估算的上限。Delphi 法^[27]是专家判断类估算法的典型代表。Delphi 法中需要一个专家小组，通过记录个人匿名提交的估算数据，并且对它们进行评审和讨论。如果不能达成共识则需要重复进行该过程直至得到最终结果。

2. 专用估算技术

(1) 功能点估算法

功能点（Function Point, FP）估算是一种软件行业专用的估算方法，适合以数据和交互处理为中心、以功能多少为主要造价制约因素的软件项目。从使用者的角度度量、而非构建者角度出发，关注于系统如何存储以及处理数据信息。功能点估算法的基本流如图 7-11，其中最主要的就是需要确定数据功能和事务功能。

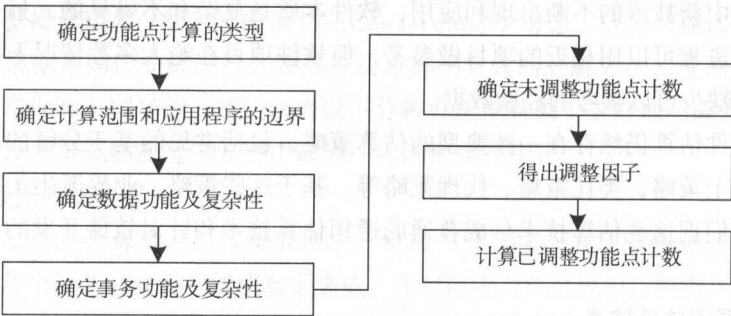


图7-11 功能点估算法流程

功能点估算中的数据功能包括内部逻辑文件（ILF, Internal Logic File）和外部接口文件（ELF, External Interface File）两大部分，而事务功能则包含外部输入（EI）、通过计算复杂结果的外部输出（EO），包括排序和聚集在内的直接输出信息的外部查询（EQ）等三大计数项。通过识别各个功能点计数项并确定各项指标系数加权求和即得到最终估算结果。

(2) 全功能点分析法

COSMIC-FFP^[28]是由通用软件度量国际协会（Common Software Measurement International Consortium, COSMIC）提出的全功能点分析方法（Full Function Point, FFP）。其基本原理是任务功能规模是通过“数据移动（Data Movement）”的个数来度量。该模型把数据移动分为 4 种类型：Entry（将数据组从用户移到目标软件），Exit（将数据组从目标软件移到用户），Read（将数据组

从存储设备移到目标软件）和 Write（将数据组移到存储设备）。其中，Entry 和 Exit 属于用户接口，Write 和 Read 属于存储器接口。每个有效的数据传送都被看成一个 CFP。CFP（COSMIC Function Point）是度量标准单位，等同于一个数据流。

COSMIC-FFP 估算过程如下：识别系统中的用例图，通过用例图确定度量范围 and 应用程序边界；识别系统中的类图，通过类图确定关注对象和数据组；通过时序图来确定用例中的有效数据移动类型和次数；识别 CFP 值并累加形成总 CFP 值。可以看到掌握 COSMIC-FFP 估算过程的前提是能够对系统进行建模并使用 UML 等工具获取用例、类和操作时序，这部分内容我们已经在 7.2.2 中做了详细介绍，从中我们也可以看到 COSMIC-FFP 确实是与软件开发过程紧密结合，是一种具有高度针对性的软件开发工作量估算技术。

7.4 质量管理

质量管理的目的在于提高客户满意度，但传统质量管理观点和现代质量管理观点存在一些差异（见表 7-2），从这些差异中我们不难看出软件行业常见的软件测试只是质量管理的一部分而不是全部。

表7-2 质量观点对比

传统质量观点	现代质量管理观点
质量是检查出来的	质量是规划出来的，而非检查出来的
质量就是指产品的质量	质量不只是产品还包括过程
缺陷是不可避免的	事情一次作对成本最低-零缺陷
质量管理是质量部门人员的事情	质量管理，人人有责
对于质量事故，基层人员负主要责任	质量责任高层管理者承担85%
质量越高越好	质量符合要求、适用、客户满意，也要考虑成本
改进质量主要靠检查和返工	改进质量考预防和评估

7.4.1 质量管理的维度

关于软件开发过程中的质量问题我们首先要明确几个概念，项目中的质量管理由三部分组成，分别是质量规划、质量保证和质量控制。质量规划（Quality Planning, QP）识别哪些质量标准适用于本项目，并确定如何满足这些标准的要求；质量保证（Quality Assurance, QA）开展经过计划的、具有系统性的质量活动，确保项目实施满足所需要的所有过程；质量控制（Quality Control, QC）监测项目的具体结果，判断它们是否符合相关质量标准，并找出如何消除不合格场景

的方法。

1. 质量规划

质量管理工作的实施与软件开发一样也需要进行计划，质量管理计划说明项目管理团队将如何把组织的质量方针付诸实践，考虑项目质量控制、质量保证和过程持续改进问题。在质量规划的过程中，质量成本（Quality Cost）是考虑的关键因素。质量成本包括一致性成本（如预防成本和评价成本）和非一致性成本（如外部失败成本和内部失败成本），软件测试就属于评价成本的一种，质量管理提倡的是尽量降低非一致性成本。

质量规划的另一个重要话题是确定过程改进计划，过程改进计划详细说明过程分析的具体步骤，以便于确定浪费和非增值活动，进而提高客户价值，通常包括确定过程边界、过程测量指标和绩效改进目标。过程改进的基本手段是使用PDCA环，通过设定为了达到目标所必需的方法或标准（Plan）→按计划逐步实施具体工作（Do）→确认并检查实施的效果（Check）→确认实际效果与计划差异并根据需要采取措施（Act）等步骤形成过程改进的闭环（见图 7-12）。

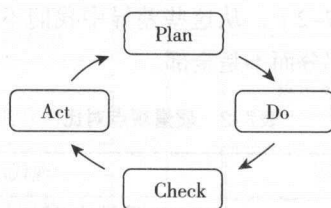


图7-12 PDCA环

2. 质量保证和控制

计划的制定是为了实施，质量管理的实施过程包括质量保证和质量控制两种类型，其中质量保证的对象是过程本身，而质量控制的对象是软件系统，这是两者的本质区别。

质量保证贯穿项目始终，往往由专门部门或组织提供，项目团队、实施组织的管理层、客户 / 发起人、未积极参与项目工作的其他项目干系人都可以是质量保证的参与者。因为质量保证关注过程，所以也可以为过程持续改进提供支持。

质量控制通过一系列工具和手段来保证软件系统的正确性，如鱼骨图（又称因果图）直观地显示潜在问题或结果与各种因素之间的联系，描述相关的各种原因以及子原因如何对质量产生潜在的影响；控制图判断如果过程处于正常范围内，就不应对其进行调整，而过程一旦失控，则必须对其调整；直方图也是常见的一种统计报告，可以显示在某个最小值和最大值之间的值的等级或范围内值出现的频率；帕累托图就是 80/20 法则的具体体现；而流程图作为最常见的工具之一，

有助于项目团队预测可能发生的质量问题，认识到潜在问题，就可以建立测试程序或处理方法。这些图同样可用于质量保证，关于它们的描述和使用方法可参考PMBOK^[9]。对于不同的场景我们可以选择不同的质量控制工具，参考表 7-3 做出相应的选择。

表7-3 质量工具与使用场景

场景	使用工具	特点
需要找出引发问题的原因	因果图	发散思维
需要判断过程是否在控制内、是否出现了典型偏差	过程控制图	按时间测量数据
需要找出影响问题的关键原因，指导采取纠正行动	帕累托图	20/80原理
需要看产品是否符合要求，可时间有限、费用有限	统计抽样	节约成本

质量管理的三大维度中包含的内容非常广泛，可以说涵盖在所有的软件开发过程中，其中关于质量保证的内容将在下一章研发过程系统建设和改进中进行详细介绍。而技术评审是软件开发过程确保开发质量的又一个重要话题，本节后续内容将围绕技术评审的实施方法展开讨论。

7.4.2 技术评审实施方法

技术评审（Technical Review，TR），其主要特点是由一组评审者按照规范的步骤对软件需求、设计、代码或其他技术文档进行仔细的检查，以找出和消除其中的缺陷。

1. 技术评审概述

技术评审的对象并不仅仅是功能也包括过程，也就是说技术评审同时面向质量控制和质量保证，一方面强调发现软件在功能、逻辑、实现上的错误并验证软件是否符合需求规格；同时也确认软件符合预先定义的开发规范和标准、保证软件在统一的模式下进行开发。

技术评审表现为一种阶段性活动。围绕一个产品或项目的开发过程，一般都会经历可行性分析、计划、开发、测试、上线等多个阶段，从严格意义上讲，每一个阶段都应该进行评审。在可行性分析阶段，技术评审可以理解为一种概念评审；在计划阶段，则需要对产品需求和开发计划进行确认；在开发阶段，主要强调开发工作的实行情况，包括对开发资源、阶段性里程碑等的确认；在测试阶段，技术评审的目的在于明确是否完成测试工作，包括产品功能测试，以及所需开展的运营、支持等方面的确认；最后的上线阶段，技术评审将证实市场推广、销售

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

支持等工作是否就绪。图 7-13 展示了在集成产品开发（Integrated Product Development，IPD）模式下各个阶段所开展的技术评审名称以及产出。

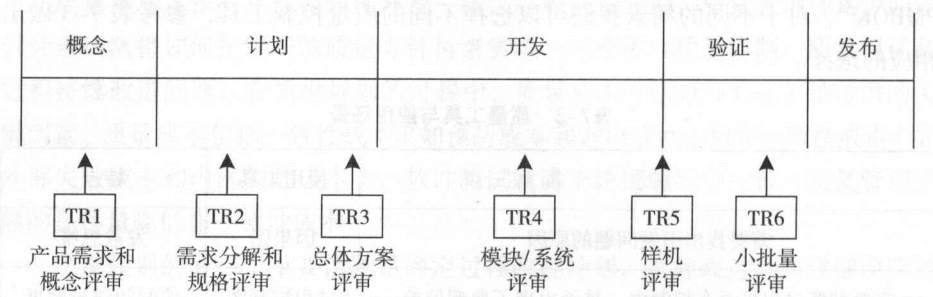


图7-13 IPD技术评审与阶段

另一方面，技术评审开展过程中也具有层次化的需求。狭义的技术评审是指正式技术评审（Formal Technical Review，FTR），也就是通常所说的同行审查技术（Peer Review）。而广义上的技术评审可以包括子评审、内部评审、二次评审等。子评审也是一种正式评审，由项目组组织，对大型产品开发流程中的子过程活动输出质量把关。而内部评审则是一种非正式的评审方式，偏向于同行设计和问题讨论。二次评审相当于预审，在大面积正式评审之前，通过小范围的评审发现大多数问题，从而减少正式评审的成本，提高评审效率。

2. 技术评审注意点

技术评审最常见的方式就是召开评审会议。评审会议的召开方式大同小异，可以根据评审需求具体展开，但存在一些典型问题点需要引起技术管理者的注意。

· 缺少必要的评审

在图 7-13 中，我们已经看到在不同的产品开发阶段需要不同的技术评审，每个技术评审主要关注产品的一个方面。尽管我们可以通过过程裁剪弱化乃至取消部分技术评审，但缺少必要的技术评审是质量问题产生的一个源头。评审过于仓促，或者可有可无，保证质量就需要过分依赖测试。确保该有的评审在执行过程中都应该要有。

· 产品需求没有二次评审

建议在产品需求中实施二次评审。为了快速适应市场需求，软件开发周期越来越短，产品经理在没有完全梳理清楚产品需求的情况下就召开需求评审会议，导致后期开发过程中出现很多问题，返工的现象比较严重。这种情况在互联网行业尤为常见。二次评审意味着在面向所有开发人员的正式需求评审之前，先可以找部分技术代表作第一次评审，针对第一次评审中需求存在的问题先修订之后再召集所有人开第二次评审。

· 评审节奏无法合理控制

评审会议也是会议，会议就需要聚焦议题。很多评审效率不高就在于会议本身有问题，典型的包括输入输出不明确、缺少主持人或主持人不善于引导、会议不是结果导向也就无法形成有效决策、会议议程空泛而不能收敛、会议虽然能达成一致但没有具体工作安排和责任人制度、即使有工作安排但缺乏跟踪和监控机制、会议相关的资料没有充分准备也没有提前交付到参会人员等。具有上述特点的评审会议很大程度上不会有实质性的成果，开完一次之后还需要开第二次，如果把握不好浪费的不但是时间还有团队的气氛，需要进行分析 and 识别。

7.5 风险管理

所谓风险(Risk)，是指一种不确定的事件或状况。在项目中，时间、费用、范围、质量等因素都可能成为风险，因为任何项目中都会存在不确定性。风险一旦发生，会产生一项或多项影响，但所产生的影响会随项目生命周期而变化。虽然风险管理并不像计划、质量管理与我们每天的工作都息息相关，但对于软件开发而言、特别是快速迭代模式下的软件开发，风险管理的重要性并不低于其他几个项目管理维度。是否具有风险意识以及应对风险的能力是判断一个技术管理者是否合格的重要依据之一。

7.5.1 通用风险管理框架

关于风险管理，业界也总结出了应对风险的通用做法，表现为图 7-14 中的风险管理过程框架。

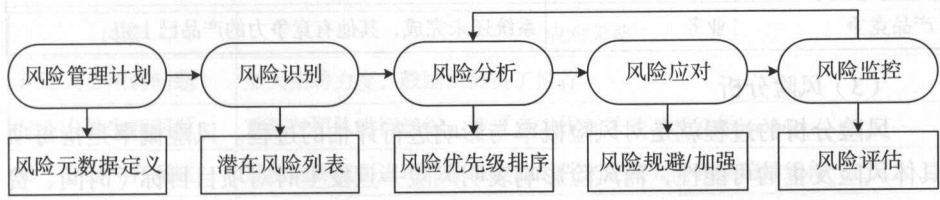


图7-14 风险管理过程框架

(1) 风险管理计划

风险管理是一项复杂度和专业性比较高的工作，建议设立相应的角色与职责进行专门负责，这些角色和职责在风险管理计划中得到明确和授权。同样，风险管理也需要成本投入，预算和时间安排是对管理工作的规划。在软件行业，风险类别体现在技术、人员、组织、工具、需求等多个方面，风险管理计划中通常需

要根据各个团队的特点对这些风险类型的划分和主要应对方法做出阐述，并给出风险概率和影响的定义，即提供风险大小的判断依据。

(2) 风险识别

SWOT 分析是识别风险的一种有效手段，通过内部资源、外部环境有机结合起来清晰地确定被分析对象的资源优势和缺陷，了解所面临的机会和挑战，从而识别潜在的风险。SWOT 分析是一种态势分析方法，通常用于对团队战略与战术等高层面风险的识别。关于 SWOT 以及各个维度之间的组合策略可以参考 2.1.3 节中的讨论。

更常见的风险识别则来自于风险类型，如表 7-4 中列举了软件开发过程中具有代表性的一些风险。这些风险一旦被识别，就需要记录到专门的风险登记册(Risk Register)，风险登记册对每项风险做唯一标识，并添加识别人、潜在原因、潜在对策、可能后果等元数据。

表7-4 风险类型示例

风险	风险类型	描述
开发人员流失	项目	有经验的开发人员会在项目进行过程中跳槽
组织架构调整	项目	项目管理层发生变化，管理人员思维方式不同
硬件缺失	项目	项目所需的基础硬件没有按时到位
需求变更	项目和产品	软件需求与原定计划相比有明显变化
系统集成延迟	项目和产品	第三方接口的开发未按时完成
估算不准	项目和产品	低估系统的开发规模
技术变更	技术	系统的基础技术被新技术取代
产品竞争	业务	系统还未完成，其他有竞争力的产品已上市

(3) 风险分析

风险分析的过程就是对风险概率与影响进行评估的过程。风险概率是指每项具体风险发生的可能性，而风险影响表明风险一旦发生时对项目目标（时间、费用、范围、质量等）的潜在影响。风险概率与风险影响可以用极高、高、中、低、极低等定性术语加以描述。

风险影响级别判定方法属于团队的过程资产，每个团队可以根据需要确立对各个项目目标的风险影响级别。当我们把风险概率和影响的标度结合起来，并建立一个对风险或风险情况的评定等级就得到了概率影响矩阵（Probability Impact Matrix）这一风险管理领域非常重要的工具。表 7-5 就是概率影响矩阵的一种表现形式，处于该矩阵左下角部分的是安全线之内的风险，而右上角部分则是重点

需要关注的风险。

表7-5 概率影响矩阵示例

概率	风险值=概率（P）*影响（I）				
0.9	0.05	0.09	0.18	0.36	0.72
0.7	0.04	0.07	0.14	0.28	0.56
0.5	0.03	0.05	0.10	0.20	0.40
0.3	0.02	0.03	0.06	0.12	0.24
0.1	0.01	0.01	0.02	0.04	0.08
	0.05	0.10	0.20	0.40	0.80
	对某一项目目标（如成本、时间或范围）的影响比值				

风险分析的最后一步就是要量化风险,通过计算风险的风险预期货币值(Event Monetary Value, EMV)可以实现这一目标。EMV 即通过货币综合表示风险对项目的影响,对于特定风险事件,用 P 表示该事件发生的概率,用 V 表示其导致的结果,则 $EMV=P*V$ 。

(4) 风险应对

通过识别得到的风险既包括消极影响或威胁,也包括积极影响或机会。消极风险或威胁的应对策略包括回避、转嫁、外包、减轻等措施,其目的就是消除风险的影响,如果不能消除则尽量减弱或通过财务手段把风险转嫁给第三方,表 7-6 是软件行业常见的消极风险应对示例。而开拓、分享、提高等则是积极风险或机会的应对策略,尽量提高风险发生的概率或使风险的正面影响因素最大化。

表7-6 风险应对示例

风险	应对策略
开发人员招聘问题	加大招聘力度,鼓励内部员工推荐
核心人员生病问题	重新对团队进行组织,使更多工作有重叠,相互做backup
第三方组件有问题	重新选择或买进可靠性稳定的组件来更换有潜在缺陷的组件
需求变更	导出可追溯信息来评估需求变更影响,扩大隐藏在设计中的信息
组织架构调整	拟一份报告提交高级管理层,说明项目的愿景和重要性
开发估算问题	用更科学的系统工程方法和相关自动化工具进行检查和确认

(5) 风险监控

风险监控工作包括识别、分析和计划新生风险,并追踪已识别风险。对于现有风险,重新分析并监测应急计划的触发条件;对于已识别和新生成的风险,采取对应的应对流程。同时,定期总结风险应对计划的实施并评估其效力。

7.5.2 软件开发与风险管理

基于通用风险管理框架，我们再次梳理软件开发过程中所存在的核心风险，并提供缓解风险的方法。

1. 软件开发的核心风险

软件开发的風險普遍存在，作为技术管理者，可能你能根据你周围团队的现状列举几十种风险，我们这里要列举的是软件开发的核心风险，即绝大多数开发工作的失败都与这些风险有或多或少的关联。

（1）不合理的开发计划

软件开发过程中处于首位的风险是不合理的开发计划，这一点几乎无可争议。回想我们在 7.3 节讨论的开发计划管理，在通用计划管理活动框架中的每一个活动都可能存在不合理的结果，更加不要说由于开发范围分解和开发工作量估算所导致的误差和错误。

开发计划中存在的误差和错误是致命的，因为这是整个开发过程开展的基础。不合理的开发计划的制定，要不是技术团队，尤其是技术管理者自身对开发任务存在理解不充分或判断失误的原因，要不就是上层管理者对计划的制定起到了决策作用。无论是哪个原因，对开发最终能否按时完成都是一个极大的风险，据研究表明^[29]，不合理的开发计划所引起的风险累积可能使项目交付延期 80% 之多。

（2）不可控的需求管理

从项目管理角度讲，需求总是在不断变化，如果不加管理就会变成不可控，我们已经在 7.2.1 中介绍了需求管理的基本理念。对于需求的变化，技术人员的立场应该是坚定的：如果你需要添加什么功能，可以，但我们的开发过程也需要进行相应的调整，并根据工作量重新制定开发计划并允许一定程度上的变化。显然，技术人员的这种立场能够减低开发过程的风险，但不一定会降低产品交付的风险，所以在现实世界里不大可能出现给开发者足够时间弹性的需求变化。最可能的一种结局是开发范围变了，但时间不可能有相应的变化，然后自然而然就产生了风险。

（3）不稳定的开发团队

在当下的互联网行业，开发人员频繁跳槽已经不是什么新鲜事。在研发过程体系和过程资产建设不是很健全的团队下，人员流失是一种高风险，而且这种风险一般很难在计划评估的时候被纳入要考虑的风险范围之内，也不大会有团队会为一个可能但又不确定是否会离职的人安排备份人员。如果离职的是一名核心人员，那么他带走的可能是整个系统的架构设计、整个团队的管理思维以及整个产品的发展愿景，很难在短时间内找到合适的人填补空缺。

（4）不可靠的开发效率

开发人员之间的工作效率存在较大差异^[30]，作为技术管理者而言，应该把团队作为一个整体来考虑，但更为重要的是，一个团队中务必需要存在一到两名核心开发人员，如果大家的技术水平没有层次，除非所有人水平都很高，不然也不利于提升团队的整体水平。一个团队的开发效率在很大程度上也受前面所列几种风险的影响。

2. 缓解风险的方法

缓解风险的方法有很多，最基本也最重要的一点就是要做到“提前”，除此之外，迭代式开发和增量式交付都有助于发现和降低开发风险。

（1）提前

缓解风险的最好办法就是将各种能做的事情都提前来做。风险本质上是一种带来不良影响的可能性，“提前”能够缓解风险的原理就在于把这种风险尽早变成现实，那么风险也就无所谓风险。当然，提前开始需要环境和各种资源的支持，一般也无法做到提前很多。技术管理者之所以能成为管理者，就在于他的思想和行动永远都要比下面的开发人员快半拍，在别人还没有意识到要去做风险分析的时候已经明白可能会发生什么，从而防止某些风险的发生或者说降低风险发生时的影响。

（2）迭代式开发

风险来自于计划、需求、团队和开发效率，所有这些相关的风险都是不可避免的，那是否有办法在开发过程上对风险进行缓解乃至扼杀呢？答案就是把过程进行缩小化管理，即把一个大而全的产品开发过程划分成一个个小的子过程，每个子过程的产出是上一个子过程的累加。每个子过程因为涉及的范围、时间、资源都比较少，那么由于计划、需求和团队上存在的风险所带来的影响也就越小。这就是迭代式开发的核心思想。

试想在一个需要 10 个人开发 3 个月的系统中，如果一开始没有采用迭代式开发，一般的做法是把系统拆分成几个大模块，然后分别对每个模块分配相应的开发资源并制定开发计划。因为所有模块都已启动开发，一旦需求变动就会涉及多个模块，一旦人员流失也势必会导致他所参与或负责的模块无法正常开发工作。如果采用迭代开发，那么在某一个迭代中，同时并行开发的可能只有一个模块，需求和人员的变动只会影响到当前迭代中的内容。当一个迭代完成，我们就可能重新评估需求和安排人员开始新的迭代。图 7-15 展示了使用迭代式开发缓解风险的原理。关于迭代式开发以及代表性的开发模型我们将在下一章中做详细介绍。

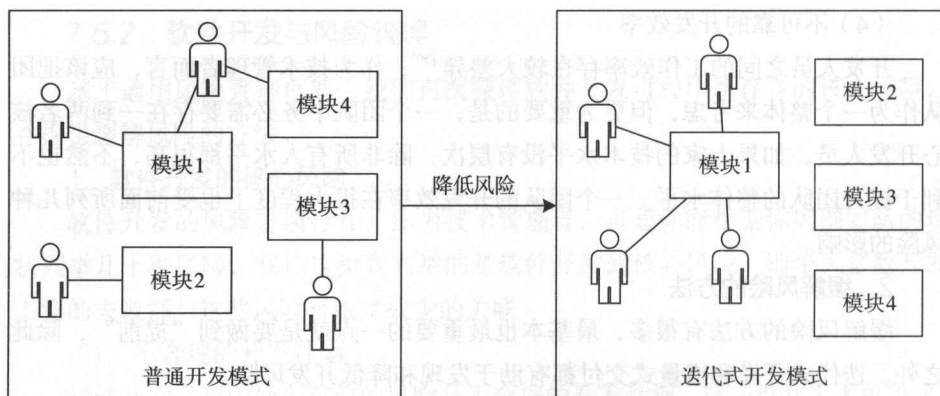


图7-15 迭代式开发缓解风险的原理

(3) 持续交付

缓解风险的另一个思路是持续交付。持续交付强调尽早发现问题，并通过自动化、配置管理、持续集成等手段降低缺陷进入下一环节的概率，从项目管理角度讲也意味着节约成本和降低风险，并提高了项目的可见性。关于如何实现持续交付就是我们接下去要讨论的内容。

7.6 交付管理

当我们完成代码的开发，从代码提交开始到最后的发布，不同的团队可能会有很多步骤。比较典型的软件交付工作流程可以抽象为代码提交→代码编译→自动化测试→手工测试→部署→预发布→正式发布，这个过程中每一步都可能出错。为了降低乃至杜绝这其中每一步的出错概率，在整个项目管理体系中就需要引入交付管理的概念，同时，交付管理也是对技术管理体系的一项补充。

7.6.1 软件交付模型概述

针对软件交付基本步骤中的出错情况，我们做一下原因分析，会发现大多数错误的发生一方面是因为没有尽早发现问题，另一方面则来源于手工配置和手工部署。通常，系统上线的第一次发布因为周期很长，各个团队之间没有形成体系化工作规范所造成的协作成本和出错概率会比较高，导致很多潜在的问题并没有在发布流程中较早的暴露出来。有时候，我们也会提出类似的抱怨说为什么试运行环境正确，生产环境却有问题。在复杂环境下，配置管理混乱、缺少必要的自动化等因素会导致手工操作的结果不可控。

基于以上分析，软件交付基本思路就是要做到自动化发布。自动化代表可重

复，并注重过程，过程对则结果一定对。在自动化发布环境下，无论什么修改都应该触发相应的监控流程，确保问题在第一时间被暴露和解决。一旦建立起自动化发布平台，就可以通过频繁发布降低出错所引起的风险，而频繁发布能够促进快速反馈，从而推动过程改进。

为了实现自动化发布，软件交付存在一些共性原则，包括为软件发布创建可重复的过程、把所有可能改变的东西都纳入版本控制、提前并频繁的进行发布演练。同时，交付过程被认为是开发、测试、运维、产品等全员的责任。DevOps（Development+Operations）就是这种思想的体现，DevOps 是一组过程、方法与系统的统称，用于促进开发、技术运营和质量保障部门之间的沟通、协作与整合。结合下一章中将要提到的敏捷和精益思想，我们可以把 DevOps 与其他方法论以及各个角色之间的关系描绘成图 7-16，其中的持续集成部分将在本章后续内容中介绍。

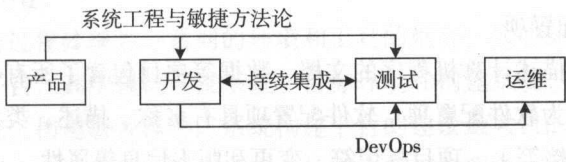


图7-16 DevOps与其他方法论和角色

围绕着软件交付模型和需求，我们抛出几个问题：

- 如何进行版本控制？
- 如何建立统一发布过程？
- 如何进行有效协作？

本节将围绕这些问题进行讨论，我们的基本思路是关注自动化与集成，并基于原则导出工程实践。要想实现自动化，首先要从配置管理入手，掌握配置管理的相关模式和工具。

7.6.2 配置管理

配置管理（Configuration Management）概念的提出在于现代软件开发过程中复杂度的不断提升。软件配置为如何处理软件开发复杂度提供了一种思路。

1. 配置管理概述

软件配置由在软件工程过程中产生的所有信息项构成，它可以看作该软件的具体形态在某一时刻的瞬间影像，而软件配置管理（Software Configuration Management，SCM）就是对这些形态和影像的管理过程。

配置管理中存在一批基本元素，在具体介绍配置管理的具体活动之前，我们

有必要了解这些基本元素（见表 7-7）并对其中几个重要元素进行展开。

表7-7 配置管理基本元素

元素	描述
软件配置项（SCI）	与配置控制下的软件项目有关的任何事物
版本（Version）	配置项的一个实例
基线（Baseline）	组成系统的各个组件版本的集合，不能改变
配置管理数据库（CMDB）	保存配置项及其关系的数据库
主线（Mainline）	代表系统不同版本的基线的序列
发布版本	发布给客户使用的系统版本
工作区间（Workspace）	一个私有的工作区间，在其中可以修改而不至于影响其他会修改软件的开发者
分支（Branch）	从现存的代码线版本中构建一个新的代码线
系统构建	通过链接组件和库的合适版本创建一个可执行的系统版本

（1）软件配置项

软件程序、描述计算机程序的文档、数据等项目包含了所有在软件过程中产生的信息，总称为软件配置项。软件配置项具有名称、描述、类型（模型元素、程序、数据、文档等）、项目标识符、变更和版本信息等属性。

（2）基线

所谓基线是指已经通过正式评审和批准的软件规格说明或代码，它可以作为进一步开发的基础，并且只有通过正式的变更规程才能进行修改。在软件配置项成为基线之前，可以迅速而随意地进行变更，一旦成为基线，变更时就需要遵循正式的评审流程才可以变更。因此，基线可看作是软件开发过程中的里程碑。围绕基线所开展的工作流程可参考图 7-17。

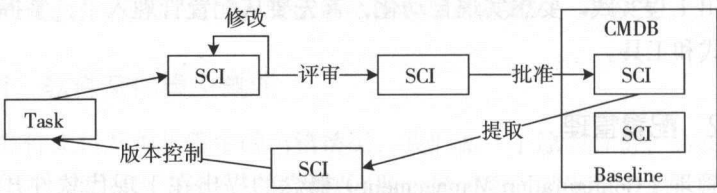


图7-17 基线流程

（3）CMDB

图 7-17 中的 CMDB 是指配置管理数据库（Configuration Management Database），是用于保存与软件相关的所有配置项的信息以及配置项之间关系的数据库，包括每个配置项及其版本号、变更可能会影响到的配置项、配置项的变更路线及轨迹、与配置项有关的变更内容、计划升级、替换或弃用的配置项、不同配置项

之间的关系等。

配置管理由一系列的活动组成，典型的配置管理活动包括系统构建（System Build）、版本管理（Version Management）、变更管理（Change Management）和发布管理（Release Management）四大部分（见图 7-18），这些活动都与上述配置管理基本元素有关，并在特定场景发挥作用。

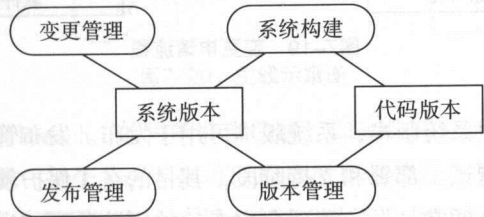


图7-18 配置管理活动

(1) 系统构建

系统构建的过程体现为一系列的环境和工具的整合，这些环境和工具构成完整的系统构建平台，其中包括开发系统、目标环境和构建系统，而构建系统又包括版本控制系统、构建服务器等。系统构建平台的建设最大目的是希望实现自动化，即把源代码、配置文件、数据文件、外部库等配置项通过编译器工具自动生成可执行目标系统，同时对该目标系统可进行自动化测试并给出测试结果。

(2) 版本管理

版本实际上体现为包含一组配置项的快照。所谓快照（Snapshot），就是系统在某一个时刻的状态，假如纳入配置管理的文件 1、文件 2 和文件 3 在不同时间具有不同版本号，而 Version1 和 Version2 就是这三个文件的不同版本号的组合，即代表它们在特定时间点的快照信息。

版本控制系统记录配置管理中所有配置项的变动，并为每次变动自动加上版本号。围绕版本控制系统，每个私人工作区间可以执行 CheckOut 和 CheckIn 操作，确保获取他人的更新以及将自己的更新提交到版本控制系统供他人使用。

(3) 变更管理

变更管理的对象是变更请求（Change Request，CR），对于一个项目而言，变更请求通常来自客户并由项目管理团队进行确认。组织级别可以存在一个 CCB（Change Control Board，变更控制委员会），通过项目管理团队确认的变更请求将提交到 CCB 中进行评估，评估的依据一方面来源于开发团队对该变更请求进行分析的结果，另一方面 CCB 中的产品开发专家也会给出其专家意见。围绕变更请求的整个变更管理流程参考图 7-19。变更管理的考虑因素包括不做变更会引起的后果、变更的益处、变更影响的用户数、变更所花成本、产品发布循环等。

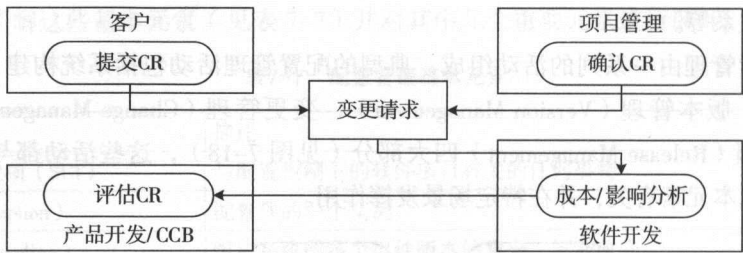


图7-19 变更申请流程

（4）发布管理

系统构建将产生系统版本，系统版本可用于发布。发布管理是指控制和监督软件发布的启动、测试、部署和支持阶段，其目的在于保护软件的生产环境和使用正确的系统服务。通常，发布管理会由专门的团队负责，生产环境也不建议直接开放给开发人员。围绕发布管理，也可以使用一系列的工具和脚本构建自动化的执行过程。

2. 配置管理模式

配置管理是一种基础性方法论，重在提供相关的思想和工具，而对如何应用这些思想和工具并没有给出明确的说明。每个团队中都可能根据自身情况提炼出相应的工作模式和最佳实践，这些模式和实践的提炼通常有两个主要的切入点，即配置管理服务器和个人工作区间，这也构成了配置管理的两个主要维度。

（1）配置管理服务器

配置管理服务器代表的是一种中央仓库，站在中央仓库的角度看，配置管理中我们可以总结以下主线模式、发布线模式和任务分支模式。

主线（Mainline）模式的提出跟代码合并（merge）的复杂性有关。我们知道当多个团队分别从服务器中获取代码版本，然后对该代码版本进行修改并试图提交时就需要合并代码。如果团队数量较多，代码变动的版本远落后于服务器上最新版本，合并所产生的各种冲突就会成为开发人员的噩梦。避免版本树太多太深，尽量减少合并，避免临时改动过度传播以简化代码同步成本是我们应对合并复杂性的基本思路。为此，对于某一个系统或模型，我们都应该在配置服务器上创建一个主线，随着各个功能的版本发布，主线的版本也随之不断演进，确保各个团队能基于某一个主线版本展开代码开发工作，并将完成的代码提交到主线以不断推进主线的版本演进。主线与各个功能发布之间的演进关系可参考图 7-20。

从图 7-20 中，我们可以引申出另外一个非常重要的概念，即发布线（Release Line）。每一次正式的发布都应该存在一个发布版本，而围绕这个发布版本可能又会发现 bug 并进行修复等情况，因此为每次发布单独确立一条发布线是一项推荐的工程实践。

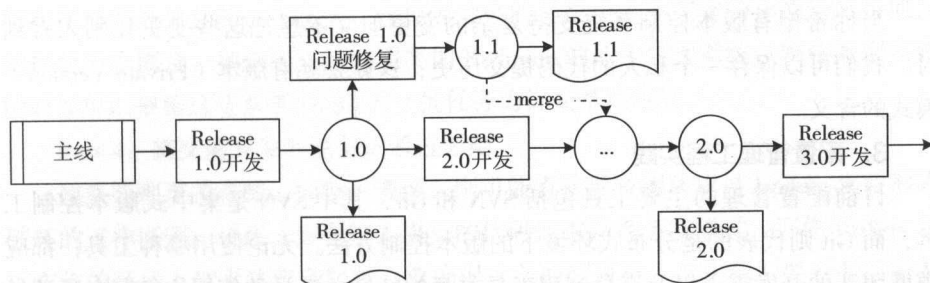


图7-20 主线示意图

任务分支（Task Branch）的目的非常明确，即确保开发版本之间的代码隔离，每个团队、每个功能、每个 bug 等都可以根据需求创建任务分支，当开发任务结束之后，分支代码合并到开发主线时即代表该分支生命周期的结束（见图 7-21）。

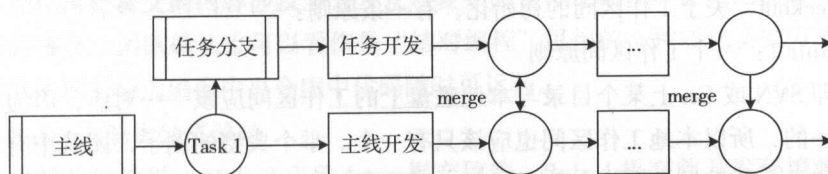


图7-21 任务分支示意图

（2）个人工作区间

开发过程中，如何确保你的工作不受外部环境影响但又能与代码线保持一致，我们就需要用到个人工作区间的概念，隔离工作区间去控制变化和更新工作区间去保持同步是我们使用个人工作区间的基本思路。个人工作区间相关模式包括仓库模式、系统构建模式和私有版本模式。

仓库（Repository）模式告诉我们要确保所有的代码、文档、脚本、第三方库等都保存在具有版本控制功能的仓库中。如果我们不知道哪些是不可以放到配置管理服务器上的，那就把它们都放上去吧。该条模式也可以归为是配置管理服务器中的内容，我们放在这里的目的在于强调开发人员应该有版本控制意识。

系统构建（System Build）模式包含两层含义，一层是私人系统构建，另一层是集成构建。如何确保你的变更不会影响到整体系统的构建？确保在提交代码之前在个人工作区间先构建成功；如何确保代码库中代码永远能进行可靠的构建？那就进行集中化构建。

系统构建与代码提交的方式和频率有很大关系，我们崇尚任务级别提交（Task Level Commit）。关于提交我们有两个问题，在两次提交之间你应该做多少开发工作？答案是不要太多；提交的次数应该如何把握？答案是能多则多。在日常开发过程中，提交小粒度开发任务被认为是最佳实践。

当你希望有版本控制系统支持复杂的变更但又不想把这些变更让别人看到时，我们可以保存一个私人的代码提交历史，这就是私有版本（Private Version）模式的含义。

3. 配置管理工程实践

目前配置管理的主要工具包括 SVN 和 Git，其中 SVN 是集中式版本控制工具，而 Git 则代表的是分布式环境下的版本控制方法。无论使用哪种工具，都应遵循团队的开发需求以及充分利用工具本身的特性。工具的使用往往存在所谓的最佳实践，对于配置管理而言，我们也可以梳理一些简单通用但又非常有效的工程实践。

（1）Checkout

Checkout 的作用是工作区间的初始化，使用客户端工具时通常从根目录全递归 Checkout，关于工作区间的初始化，有一条原则：

Rule 1：一个工作区间原则

即 SVN 或 Git 上某个目录与本地磁盘上的工作区间应该一一对应，因为目录是唯一的，所以本地工作区间也应该只有一个。举个典型的例子，团队中有些成员会把一个项目目录下的产品文档 Checkout 到 C 盘，项目文档 Checkout 到 D 盘，然后把代码放到 Eclipse 的 workspace 里，这样大家一讨论问题，在各种目录之间切换，有时候因为没有对多个本地副本同时进行 Update，导致本地存在多个不同版本的文档、代码，沟通上就不在同一层面，影响团队协作效率。关于 Checkout，还有一个常用的功能就是工作区间镜像，以便查看历史版本以定位问题、查看历史信息。

（2）Update

Update 操作的对象是本地的的工作区间，该操作的核心原则有：

Rule 2：初始化工作前更新原则

即在开展与工作区间相关的工作前，确保执行 Update 操作，Update 操作的对象可以不是整个工作区间，但必须至少是本次工作相关的目录层次。初始化工作的时机通常有开展具体工作时，如开始写代码 / 文档；获取更新通知时，如收到邮件通知说有文件更新，或团队交流开始之前。

（3）Commit

相较 Checkout 和 Update 以本地工作区间为操作对象，Commit 操作面向的是整个中央仓库，对团队协作影响重大，有以下几条原则需要遵循：

Rule 3：完整 Issue 提交原则

这里 Issue 泛指一个独立完整的开发单元，实际情况下可能包括 Feature（功能点）、Task（开发任务）、Bug/Defect（系统缺陷）。这条原则的宗旨在于确保

每次提交都是独立的、隔离的，避免因为你的提交引入不完整的功能从而与他人的提交产生耦合。如果团队在使用类如 Redmine 的任务管理系统，则该原则实施的时候可以根据这些系统中的某个 / 批任务为提交基本单元。

Rule 4: 提交前开发环境测试通过原则

这条原则至关重要，如果不遵循，则可能直接导致团队成员 Update 之后形成破坏的工作区间。试想，如果别人提交的代码连编译都不能通过，那你 Update 之后在你的环境上就无法启动服务。要确保这条原则可行，首先我们要有专门的开发环境；其次如果你的团队在实施单元测试，则需要提交前保证单元测试全部通过；再次确保执行基本的冒烟测试以保证这次提交的版本不存在服务启动和运行上的问题。

Rule 5: 协作环境下复杂业务逻辑提交前确认原则

当你需要提交的内容涉及多人开发、复杂的业务逻辑，那提交之前最好与团队进行确认。团队确认也可以看作是“结对编程”思想的一种弱化实施方式，站在团队协作的角度确保中央仓库中代码随时可运行。

Rule 6: 职责分离原则

虽然我们遵循了 Rule 3 完整 Issue 提交原则、Rule 4 提交前开发环境测试通过原则和 Rule 5 协作环境下复杂业务逻辑提交前确认原则，但提交导致更新冲突还是不可避免。既然冲突不能避免，那我们就想办法减小冲突发生的概率，有两点思路，其中一点就是职责分离，具体到操作层面可以把系统按照模块 -> 功能线 -> 功能点的角度进行划分，然后尽量安排开发人员单独负责其中的模块或功能线，把能够分离的功能进行独立管理和维护；另一点思路就是最小粒度频繁提交原则，见 Rule 7。

Rule 7: 最小粒度频繁提交原则

关于最小粒度，可以参考 Rule 3 中的表述，一个独立的不可分割的 Feature、Task、Bug/Defect 就是能够提交的最小粒度。至于频繁提交，就是尽量保证中央仓库中的代码与团队成员中的本地工作区间同步，所以该原则也包含频繁更新的含义。如果提交和更新频率够高，那就算发生冲突，冲突影响的范围以及修复成本通常都不会太高，最麻烦的莫过于长时间代码不同步导致各种工具不能自动解决的冲突问题，这也是创建分支之后进行代码合并操作中通常要面临的问题。

Rule 8: 本地自动生成文件不要提交原则

像 Eclipse、IDEA 等工具会根据 SVN 上代码生成很多辅助性文件，这些文件通常跟本机环境相关，如果把这些环境相关文件进行提交则 Update 之后就会产生更新提示，影响团队成员的代码管理。通过 TortoiseSVN 等客户端工具，我们可以

把这些文件添加到忽略列表（Ignore List）中，这样提交时系统将自动识别出哪些文件需要提交而哪些文件不需要提交。

Rule 9：提交添加日志原则

提交添加日志原则即在提交时包含本次提交的日志，方便后续的跟踪和版本对照。添加提交日志的方法建议：日志信息主要记录的是每次的修改内容，把一些重要数据、关键操作写到日志信息中；涉及 Redmine 等任务系统上相关 Issue 修复，添加 Issue 号；修改人和提交时间由软件自动记录，无须人工写入日志信息。

本节中梳理的 9 条原则贯穿着团队中每一个成员的日常工作，我们也可以看到部分原则与配置管理中的模式完全一致，如 Rule 4 和 Rule 7 对应于系统构建模式。对于这些原则，可以根据实际情况作对应的裁剪并在团队甚至是组织级别基本达成一致。对于 SVN 和 Git 等工具，这些原则具有很大程度上的适用性。

7.6.3 持续交付

1. 持续集成

持续集成强调尽早发现问题，降低缺陷进入下一环节的概率，从项目管理角度讲也意味着节约成本和降低风险，并提高了项目的可见性。对于开发团队而言，持续集成理念的引入可以帮忙团队成员培养一种意识，建立团队对开发产品的信心。

（1）持续集成元素

· 多种角色

实现持续集成需要多种角色参与其中，也能切实解决这些角色所面临的一些问题。通过持续集成就能够把开发人员的部分重复性工作自动化，控制和降低 bug 率并实现按需发布版本。开发、测试和运维也恰恰构成了 DevOps 中的三个维度，从这个角度讲持续集成是 DevOps 的一种表现形式。

· 版本库

带有版本控制功能的中央仓库是实现持续集成的基础，关于版本库的工具和实践我们已经在上一节配置管理中做了全面介绍。

· 构建脚本

自动化是我们的目标，实现自动化的基本手段就是通过各种构建脚本把原本需要手工执行的步骤转变为系统自动执行。通常，构建脚本的目的在于集成各种第三方工具并通过一定的策略使这些工具能够相互协作。

· 持续集成服务器

构建脚本的集合实际上就可以称为构建服务器，但开发一套功能强大、用户

体验好的集成服务器成本太高，所以我们一般会使用业界主流的工具作为我们的主服务器，这些持续集成服务器都提供了较高的可扩展性，可以通过编写部分构建脚本并嵌入其中实现集成的定制化需求。

• 反馈

反馈即通过一系列的监控机制确保集成过程中每一个步骤都能进行审查和确认，并提供邮件、IM 等一系列反馈机制确保尽早发现问题并解决问题，这点同样与软件交付模型的目标相一致。

(2) 实现持续集成

引入持续集成的过程也是一个循序渐进的过程，在没有任何持续集成经验的团队中，尝试从新系统开始是一个不错的选择，对于遗留系统则倾向逐步过渡。但不管是新老系统，当项目开始时就该采取行动并开展多角色协作，最好有专人负责持续集成服务器中的追踪结果。图 7-22 是引入持续集成之后的工作流程，我们可以看到，从以 Redmine 为代表的问题跟踪系统开始，伴随着全局唯一的 IssueId，任何需求和 bug 都会流转到开发人员，开发人员通过版本控制系统进行个人工作区间和中央仓库之间的交互，最终所有的版本共享通过持续集成服务器生成可交付成果。

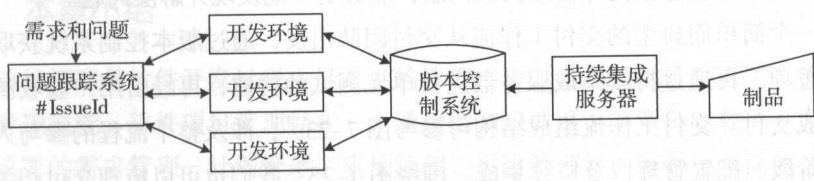


图7-22 持续集成流程

图 7-22 中问题跟踪系统和版本控制系统我们都已经有所了解，而持续集成服务器是一套完整的解决方案，能够从持续测试、持续审查、持续部署和持续反馈等维度提供支持。

持续测试包括自动化单元测试、自动化组件测试、自动化系统测试以及各种测试分类和可重复特性。持续审查通过采用组织级别的标准建立代码审查机制并持续进行设计检查，试图从减少重复代码、判断代码覆盖率、降低代码复杂度等角度为开发人员提供代码质量的可视化视图。持续部署要求随时随地发布可工作的软件，通过持续集成服务器，能够为每一个构建打上标签、执行所有测试、创建构建反馈报表，并具备回滚构建的过程能力。持续反馈提倡以正确的方式在正确的时间将正确的信息传递给正确的人。这里正确的方式可以是电子邮件、声音和 IM；正确的时间应该支持每天、每周或问题发生时；正确的信息包括构建状态、审查报表、测试结果；而正确的人通常包括系统构建人员、开发人员以及架构师。

基于持续集成流程，所有的开发人员需要在本地工作区间上做本地构建，开发人员每天至少向版本控制库中提交一次代码，开发人员每天至少需要从版本控制库中更新一次代码到本地工作区间。需要有专门的集成服务器来执行集成构建，每天要执行多次构建，每次构建都要 100% 通过，修复失败的构建是优先级最高的事情，这些构成了持续集成实现的基本原则。

持续集成相关的工具有很多，可供选择的不下数十种。本书中提到的 Redmine、Maven、SVN、Git 等都很常用，而对于专用的持续集成服务器，目前最流行的当属 Jenkins (<https://jenkins.io/>)。

2. 交付 workflow

本节围绕着配置管理和持续集成这两大主题对软件交付的思路、工具和工程实践做了讨论，配置管理和持续集成构成了交付模型的基础，两者之间也需要相互渗透才能构成完整的工作流程。交付工作流 (Delivery Workflow) 指的就是系统程序的配置、源代码、环境或数据的每个变更都会触发创建一个新工作流实例的过程。交付工作流属于技术范畴，也体现为一种管理性过程，其目标就是让团队所有人都能参与到软件构建、部署、测试和发布过程中来并确保过程对所有人可见，通过流程规范性提升信息反馈机制，能够更早期地发现问题。

一个简单而典型的交付工作流从交付团队出发，通过版本控制系统获取代码和配置项，再通过持续集成服务器进行单元测试的构建，再经由用户验收测试最后完成交付。交付工作流组成结构可参考图 7-23^[17]，涉及整个流程的参与人员、流程阶段、配置管理以及持续集成。围绕图 7-23，我们也可以梳理交付相关的最佳实践，如只生成一次二进制包、对不同环境采用同一套部署方式、采用预发布环境、对部署进行冒烟测试、每次变更都能在工作流中传递并做到一旦有环境出错就停止整个流水线。交付工作流的建设需要完备的配置管理和持续集成系统，同时需要端到端的 DevOps 管理系统，让团队每个成员都对项目成败负责。

在交付工作流的实现上，每个团队对交付过程理解不同，所设计的交付工作流自然也会有所差异。不管各个团队的过程如何，使用 Maven、Jenkins 以及 Python 等各种脚本语言构建和部署自动化是所有团队都可以采用的实践方式，同样，我们也鼓励使用基于 JUnit、Checkstyle、PMD、Findbugs 和 Sonar 等自动化单元测试和代码分析工具与 Jenkins 进行有效整合。最后也是最重要的是量化和改进过程，通过自动化测试覆盖率、代码风格验证、缺陷数量、交付速度、构建成功和失败次数、构建时间等量化指标我们可以找到相应的改进点并通过回顾等手段进行持续改进。

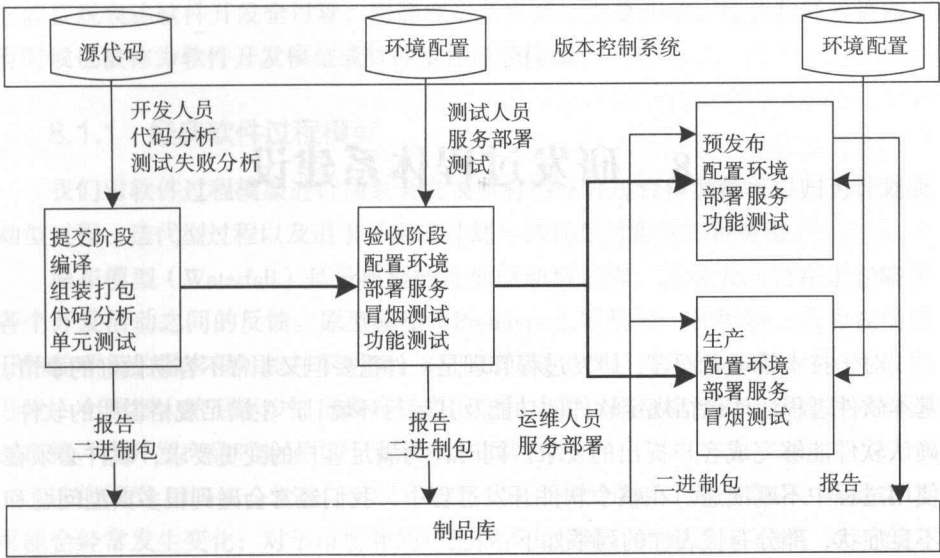


图7-23 交付 workflow 组成结构

7.7 本章小结

项目管理是软件开发过程中最常见的管理维度，也是技术管理者需要熟练掌握的管理技能。软件项目管理存在一个通用的知识体系，本章针对软件开发过程中最重要的需求管理、计划管理、质量管理、风险管理等内容做了全面阐述，并针对软件开发的特点和常见的问题给出了相应的管理方法和技巧。

本章还包括对软件交付管理的介绍，侧重于从配置管理和持续集成两个角度出发对软件交付模型展开讨论并给出了可直接应用到日常开发过程中的若干最佳实践。

8 研发过程体系建设

对于技术管理者而言，研发过程管理是一件重要但又非常不容易做好的事情。基本软件过程活动包括规定软件的功能及其运行环境、产生满足规格说明的软件、确认软件能够完成客户提出的要求。同时，为满足客户的变更要求，软件必须在使用过程中不断演进。在整个软件开发过程中，我们经常会碰到很多典型问题和不良症状，部分有代表性的列举如下：

- 研发进度不明确或难以把控
- 不同职能团队之间的衔接工作无法顺利展开
- 开发任务间的依赖关系过于复杂
- 研发团队中角色职责和分工理解不明确
- 开发活动缺少统一的组织和实施方法
- 研发团队“救火”的场景过多
- 有太多时间浪费在没有附加值的工作上

围绕上述问题，在研发过程管理领域提出了很多用于解决这些问题的过程体系。除了上一章介绍的基于 PMBOK 的项目管理体系之外，还包括 PACE、IPD、CMMI、SCRUM、XP、LEAN 等内容，这些内容涵盖了包含研发管理、团队管理、战略管理的多个方面。本章将从研发过程体系建设角度出发，有选择性的对这些模型和方法进行阐述，并分析如何将它们应用于研发过程改进。

另一方面，软件开发的特点又决定了我们不能照抄照搬业界的主流模型或方法，因为任何一种模型或方法都是基于特定的场景所设立，不同的团队、不同的研发文化，势必需要采用不同的研发过程管理方法。这就需要技术管理者能够对过程进行裁剪，开展过程资产建设和引入适合自身团队特点的过程模型。本章也会对这一主题展开探讨。

8.1 软件过程模型概述

软件过程模型（Software Process Model）是软件开发活动和任务的结构框架，

它能直观表达软件开发全过程，明确规定要完成的主要活动、任务和开发策略，有时候也被称为软件开发模型或软件生存周期模型。

8.1.1 经典软件过程模型

我们对软件过程模型进行抽象会发现所有的软件过程模型都可以归为计划驱动型过程、迭代型过程以及追求平衡的计划 - 迭代型过程这三种类型。

瀑布模型（Waterfall）是最典型的计划驱动型过程，其最大问题在于忽略了各个开发活动之间的反馈。原型模型（Prototype）则是另一种思路，认为在项目开始前项目的需求很可能并不明确，对于如何减少项目需求的不确定性，可以先开发一个原型，然后逐步细化最终形成可交付成果。原型模型对于确定显示界面或在第一次开发产品时验证可行性等场景非常有用。

迭代模型（Iterative）适合于以下场景，项目开始明确了部分需求，但是需求可能会经常发生变化；对于市场和用户把握不是很准，需要逐步了解；对于有庞大和复杂功能的系统进行功能演进，需要一步一步实施。采用迭代模型的开发过程可能表现为图 8-1。

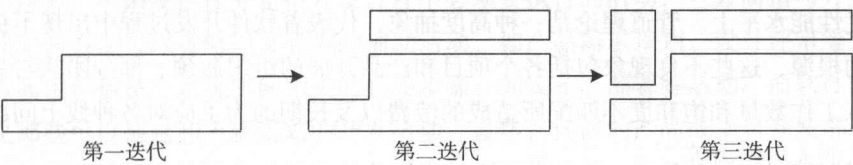


图8-1 迭代模型

以上模型中，瀑布模型是计划驱动型过程的典型代表，而原型模型和迭代模型属于迭代型过程，为了利用这两种类型的各自特点，有时候倾向于两者之间的结合。计划与迭代的结合点包括瀑布中添加原型、瀑布中添加迭代、迭代内部使用瀑布等。如图 8-2 就是瀑布 + 原型的一种实施方式。

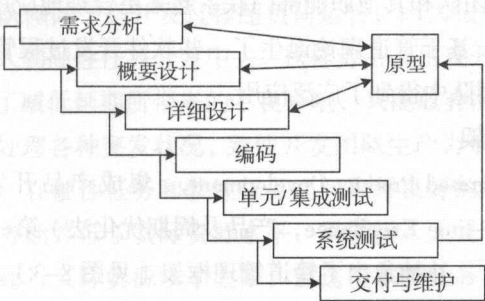


图8-2 瀑布+原型模型

螺旋模型（Spiral）是瀑布 + 迭代并考虑风险的一种混合模型。螺旋模型在整体流程上仍然是从需求到开发再到测试的瀑布过程，但在整个过程中使用迭代理念进行过程自身的演进。如第一次迭代明确系统的目标、约束和愿景，第二次迭代确定系统需求，第三次迭代中系统进行设计和开发，第四次迭代则确保系统能够进行测试。螺旋模型特别适合不确定因素和风险是主要制约因素的产品开发，如项目规模很大、采用了新技术、可能发生一些重大的变更，同时用户对自己的需求也不是很明确，需要对一些基本的概念进行验证等场景。

瀑布 - 迭代模型是迭代与瀑布的一种结合，实际上体现为一种实用主义，瀑布定义了前期项目计划过程，通常发生在技术和业务部门之间；然后采用迭代思想实现在瀑布阶段事先敲定的总体计划，迭代本身代表一个可控的非频繁交付的产品周期，通过组织级策略来监管，也受限于企业基础架构。

8.1.2 管道理论

软件开发过程本质上体现的是一种管道（Pipeline）思想，为了确保对开发的投入和产出达到平衡，必须考虑完善的资源分配策略从而确保技术开发管理处于最优性能水平上。管道理论是一种高度抽象，代表着软件开发过程中出现不良现象的根源，这些不良现象包括各个项目和产品开展的组织瓶颈、部分团队与其他团队工作数量和饱和度不匹配所造成的浪费以及长期的为了应对各种线上问题而投入的人力物力等。

上一节介绍的经典软件过程模型在一定程度上能够缓解这些问题，但在整个开发团队中跨项目、跨产品工作模式下，有一些问题需要我们进一步深入剖析：

- 如何确保不同项目和产品取得进展以及分析它们之间的相互影响
- 如何分析技术团队存在的瓶颈以便更好地促进团队协作
- 如何把技术团队和其他职能部门联系起来更合理地应用开发资源

围绕这些问题，基于管道理论诞生了一些软件开发过程管理的方法，这些方法目前在软件开发团队中得到了广泛应用。

1. 管道管理框架

作为 IPD（Integrated Product Development，集成产品开发）的溯源，PACE（Product And Cycle-time Excellence，产品及周期优化法）第一次提出了管道管理的理念和实践方法^[7]，并抽象出了管道管理框架（见图 8-3）。PACE 认为管道管理通过战略平衡、管道载量和协调部门之间的交接将重要过程与各个团队之间联系起来。这里的团队包括技术、产品、项目、测试、运维等各个职能团队。

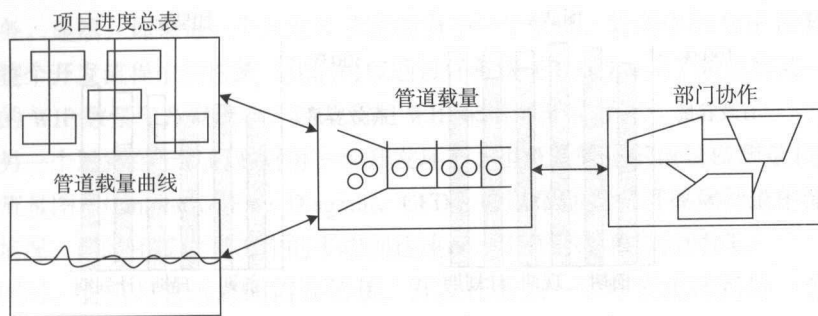


图8-3 管道管理框架图

在管道管理框架中使用了几种有用的工具，包括项目进度表、管道载量曲线图与职能部门协作方式，其中管道载量对于软件开发过程影响最大，我们将重点对其展开剖析。在开发过程中，管道载量是控制和管理技术团队与其他职能团队的依据，用以确保技术团队与职能团队之间的平衡，避免产生瓶颈现象而降低整个管道的速度。

PACE 认为，在软件开发过程中，存在两种不同性质的任务，即硬性任务和软性任务。所谓硬性任务是指开发过程中必须要执行的活动，一方面指与开发决策相关的看似可以省略但又必需的辅助活动，如产品需求规划、设计评审等；另一方面也包括整个系统实施相关的必要步骤，如发布、部署等活动。而软性任务则是那些可以拖延和重新规划的开发活动。显然，任何的产品或项目开发都是硬性任务和软性任务的结合。硬性任务通常是不可优化的，但硬性任务与软性任务的比例是优化的主要对象。在管道容量许可的前提下，通过管道的硬性任务与软性任务之间的资源比例是管理优化的关键。

我们可以使用上述硬性任务和软性任务的分类思想结合软件开发过程的特征，对两个虚拟开发团队的工作状态通过管道载量分析做一下对比。如图 8-4 所示，团队 A 中仅看硬性任务安排已经长期超出团队的负载水平，更不要说再加上软性任务。这个团队倾向于是开发过程超负荷运转，因为很少有软性任务，所以开发资源都已经投入到硬性任务开发中去。这样势必会出现开发资源跟不上现象，导致各种延期。为了减低延期所带来的不良影响，只能放弃部分质量而拼命赶工，最终整个团队疲于处理各种突发状况，致使开发团队生产力下降。而对于团队 B，通过分析管道负载，在硬性任务和软性任务之间取得良好的平衡。其表现形式就是计划内的硬性任务始终低于实际资源水平，需要投入大量资源的偏重于技术开发类的软性任务则超过实际资源水平上限，但这些软性任务可以进行重新分配和推迟处理。这种针对软性任务的弹性化处理方式既可以确保管道的顺利运作，也能保持较高的生产力。

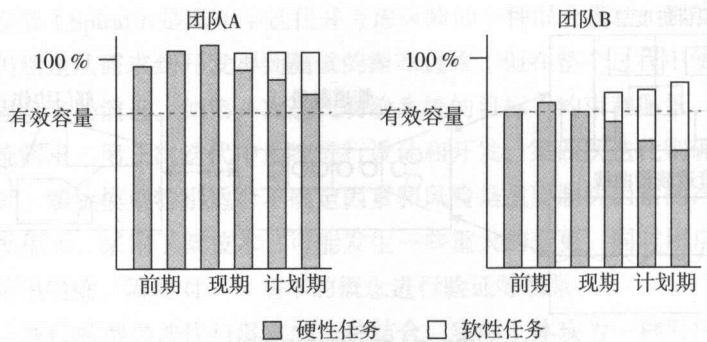


图8-4 两个团队管道载量对比图

以上对硬性任务和软性任务的分析实际上也比较直接和明确，但真正要做到管道中的合理载量并不容易。硬性任务与软性任务比例受技术开发过程中的很多因素影响，包括技术体系类型、项目管理水平、团队协作模式、决策者风格等。

对于技术本身而言，创新性技术相较成熟型技术存在较多的硬性任务，为了验证新技术的有效性和应用方式，不得不做很多探索性、验证性的开发工作，这些工作都属于硬性任务，因为这些工作没有得出合理的结果往往意味着后续的软性任务无法正常展开。

项目管理水平对开发管道的管理也会有很大影响，主要体现在资源协调上。一个团队的项目管理水平越高，一般硬性任务的比例可以相对高些，因为在开发过程中出现预料之外的可能性会比较低，也就不需要处理太多软性任务所带来的不确定性。

团队协作模式对管道负载的作用主要体现在团队组织架构上，从产品、项目、技术、测试、运维这些不同角色之间的信息流转方式和同步机制，我们可以想象团队内部的协作与跨团队之间的协作势必会同时影响硬性任务和软性任务。

决策者会对团队的开发节奏施加压力，所以其管理风格同样决定管道中硬性任务和软性任务的比例。如果决策者比较强硬且对技术缺乏必要了解，可能导致图8-4中团队A中硬性任务已经超过团队负载的情况。显然，我们希望决策者的风格能够帮助团队形成图8-4中团队B的管道负载水平。

2. 管道理论与软件开发方法

对管道以及流转于其中的硬性任务和软件任务的分析，从中得出的基本思路就是要确保两者之间维持一种比较平衡的比例。如果我们把开发工作看成是一系列环节的组合，那么每个环节实际上就是一个管道，围绕如何控制管道中的负载和流量，从软件开发角度出发存在一些思路。

首当其冲的一条思路就是限制在制品从而控制管道瓶颈。所谓在制品(Working In Process, WIP)指的就是处在某个开发环节中但还没法流转到下一个环节的开

发任务。显然，如果每一个开发环节就相当于一个管道，管道中的 WIP 的数量决定着整个开发流程中的瓶颈。我们可以通过任务板（Task Board）来观察某一个环节中的 WIP 数量和开发成本，以及利用 WIP 来增加开发缓冲。

另一个思路在于我们要对每一个开发环节，也就是管道的流量做测量和管理。累计流量图（Cumulative Flow Diagram, CFD）可以帮助我们可视化管道中流量的流转情况，同时还提供了工作的平均到达速度、工作存量和交付时间。

同时，管道与管道之间需要衔接，开发任务从一个环节流转到另一个环节势必伴随着切换成本和由于等待而导致的一定程度上的浪费。价值流图（Value Stream Mapping, VSM）的作用就在于能够让我们可视化这些浪费，从而可以分析产生的原因以及获取消除浪费的方法。

以上这些思路体现在精益开发、看板方法等不同的软件开发过程模型中，目前业界把这些过程模型与 Scrum、极限编程等一起并称为敏捷方法。下一节我们将对敏捷方法的理念以及各种代表性的方法模型做全面介绍。

8.2 敏捷方法

普遍认为 2001 年敏捷宣言的提出宣告了敏捷方法论的正式确立，敏捷宣言中强调的敏捷软件开发四个核心价值是：个体和互动高于流程和工具、工作的软件高于详尽的文档、客户合作高于合同谈判以及响应变化高于遵循计划。同时，敏捷宣言还提供了一系列的指导性原则^[10]。这些敏捷价值观和原则的背后包含着与软件开发深层次的理念。

8.2.1 敏捷的理念

结合软件开发过程，我们把敏捷的理念分成四大块内容，即适应变化、团队协作、交付价值和过程改进。

1. 适应变化

敏捷最为人称道的一个理念就是“拥抱变化”。为什么“拥抱变化”这一理念会深入人心，很大程度上是受软件开发的本质特征所决定，即任何软件的需求并不是在一开始就能完全确定。期望我们的用户、客户在开发之前想清楚真正想要的软件是不现实的，软件开发的过程就是不断向用户交付软件产品并不断激发用户或客户发现需求的过程。

逐步提炼需求的现实意味着软件开发应该以小批量、频繁交付的节奏进行，这是敏捷开发的一条基本原则，也是软件开发的客观规律。在需求响应周期相同的情况下，一次开发的需求量越小，开发资源利用率就越高；在资源利用率相同

的情况下，批量越小，其交付周期就越短。

敏捷开发的另一条重要原则是即使在开发后期，我们也欢迎需求变化。因为变化势必会发生，一开始就制作一个大而全的计划并付诸开发实施容易造成返工。对于软件开发的演进应该采用迭代思想，根据迭代累积的经验和需求变化的情况对计划做出调整。

如何适应变化的另一个思路是建立强大的反馈机制，反馈意味着提供快速响应变化的信息来源。利用多层次的反馈手段，在不断变化的环境中能够使信息透明，了解变化所带来的影响以及与变化后目标的差距，从而不断调整开发过程。

2. 团队协作

敏捷方法认为，在产品开发团队中，面对面交流是最有效的沟通方式。而沟通的最大障碍是不同职能部门的人员所具有的不同视图和视角。产品有产品的思维模式，技术也有技术的工作方法，在两者之间如果无法建立一种统一语言（Ubiquitous Language），对需求的理解就会产生偏差。所以，在整个产品开发过程中，拥有专业领域知识的业务人员和开发人员应该每天都坐在一起工作，这是敏捷方法的一条重要原则。

另一方面，敏捷团队的协作方式需要从传统型转变成敏捷型，即从垂直管理转变到水平管理。在传统方式下，管理者努力控制团队，通过制定详细的开发计划和安排，通过复杂的管理手段和规则监控开发过程。但在敏捷团队中，更多的是激励机制。以受激励的开发人员为核心构建团队，帮助团队提供资源、排除障碍，并营造自我管理的工作方式。

3. 交付价值

敏捷思想认为最优先要做的是尽早、持续地交付有价值的软件从而让客户满意，这是最重要的一条原则。敏捷方法所提倡的适应变化的思维能够为客户维持竞争优势，因为面对需求变化的第一步是尝试从客户的角度看问题。

对于软件开发而言，项目管理三角形中的时间和成本往往无法改变，在时间、成本存在冲突情况下，能够改变的就是范围。敏捷关注频繁的交付系统，且只交付刚刚好的系统，从项目的角度讲就是不要对开发范围进行镀金，对客户需求进行深入理解并简化。

4. 过程改进

开发过程中普遍存在浪费，其表现形式有很多，如部分未完成的工作、完成之后未应用的特性、由于人员更替所需要的工作交接和再次学习成本等。敏捷原则提倡尽最大可能减少不必要工作，通过识别和消除浪费来提高开发效率。

任何软件开发工作都是有时效性和阶段性的，即不存在一成不变的开发过程，也不存在完美到不需要做任何调整的过程。敏捷开发强调过程改进，通过团队定

期回顾来探寻如何提升效率的方法，并依此调整。

敏捷理念为我们提供了指导思想和原则，业界围绕这些思想和原则诞生了一批具体的实现方案和框架。本节后续内容将结合敏捷理念与管道理论对目前主流的 Scrum、精益、看板等方法做进一步阐述。

8.2.2 Scrum与过程管理

1. Scrum 简介

Scrum^[31] 基本组成结构见图 8-5，由一组称之为 Sprint 的迭代周期组成，典型的迭代周期为 1-2 周或者最多一个自然月，产品的设计、开发、测试全部都在一个迭代内完成。

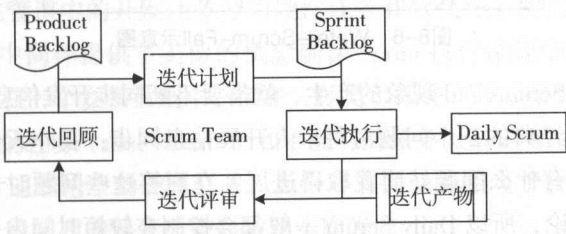


图8-5 Scrum框架

Scrum 中存在三大角色，分别是 Product Owner（简称 PO）、Scrum Master（简称 SM）和 Development Team（简称 DEV）。PO 交集比较广泛，对内需要与 SM 和 DEV 有效协作，对外对接客户、用户以及各种干系人，所以 PO 在具备领域知识的同时，还需要有较高的交际能力、决策能力和责任感；SM 的责任包括团队教练、过程执行者、外部影响的屏障、团队障碍的移除者和引入变化的主导者，每一项都需要 SM 具备专业的技能、善于发现问题、耐心并善于协作，同时 SM 作为团队主要的协调者（Facilitator），合理把握职责的边界，很多时候需要对团队透明；Scrum 中对开发团队要求较高，首先在团队成员构成上应确保跨职能，即团队包含交付一个产品所应具有的各种角色。团队内部成员之间能够进行透明化沟通，在具备对某一领域深度了解的同时还应有一定广度。

Scrum 活动构成了整个 Sprint 的工作流程，从 8-5 中可以看出 Scrum 框架中一个 Sprint 包括计划、执行、评审和回顾四大活动。

(1) Sprint 计划

Sprint 采用两阶段会议方法确定开发计划，可以分别用 What 和 How 来概括这两个阶段的目的。阶段一根据优先级以及本次迭代能完成的团队工作量填充相应的 PBI（Product Backlog Item，产品待办项），该阶段由 PO 主导。阶段二根据 PBI 展开讨论并确定实现方案，该阶段由整个开发团队主导，PO 负责解释领域问

题。Sprint 计划的产出就是 Sprint Backlog，通过计划会议，面向业务的 PBI 转变为面向开发的 Task。

(2) Sprint 执行

Sprint 执行过程中，输入是 Sprint Backlog，而输出就是可以运行的交付产物，整个 Scrum 团队都会参与其中。在 Sprint 执行过程中，需要重点关注的就是避免将一个 Scrum 迭代演变成 Water-Scrum-Fall 式迭代。所谓 Water-Scrum-Fall，指的就是在一个 Sprint 中依然采用类似分析、设计、编码、集成、测试的传统瀑布流程，导致在 Sprint 结束时，可能都还没有任何可以交付的产物（见图 8-6）。

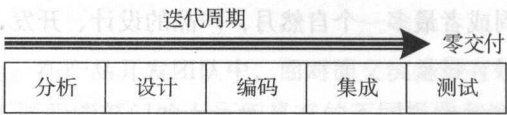


图8-6 Water-Scrum-Fall示意图

避免 Water-Scrum-Fall 现象的产生，就需要不断同步开发信息并做相应调整，Daily Scrum 通过经典的三个问题对每个人开展信息同步：我昨天做了什么？我今天将要做什么？有什么问题妨碍我取得进展？在回答这些问题时，一定要简洁，避免陷入细节讨论，所以 Daily Scrum 一般都会控制在较短时间内，对于长时间的个人发言要及时调整以控制会议节奏。

(3) Sprint 评审

Sprint 评审流程上，对于演示的结果需要进行确认，通过迭代的功能是否满意、下一个 Sprint 是否可以继续这两个问题可以明确迭代的结果。

(4) Sprint 回顾

在 Scrum 中，回顾（Retrospective）是单独一个流程节点，意味了每个 Sprint 必须完成回顾才能结束。回顾通过回顾会议展开工作，其效果就在于把 Scrum 团队中的想法全部收集起来，然后通过分类发掘，明确可以引入的变化以及丢弃当前不好的做法。回顾非常之重要，在下一节过程改进中还会有专门介绍。

2. Scrum 中的管道管理

管道思想在 Scrum 中的体现主要在对 Product Backlog 的梳理（Grooming）上，如图 8-7 所示，当原始的粗粒度需求通过管道流向开发团队，这些需求被慢慢分解和精细化。当需求到达开发团队，即形成 Spring Backlog 时，需求应当处于可进行评估和直接开发的状态。



图8-7 Product Backlog与管道

显然，管道的负载分析在图 8-7 中能发挥其作用，我们不希望出现管道两端不匹配的情况。如果原始需求梳理的速度跟不上开发团队的开发速度，管道将出现空置现象，开发团队也就无法规划和执行下一个 Sprint；反之，如果把过多的已完成梳理的需求放到管道中，那么开发团队就无法按照正常节奏完成管道中的开发任务，从而导致任务的累积。管道空置和过度累积都是比较明显的浪费现象。

针对管道负载可能出现的问题，PO 日常工作的一大重点就在于梳理 Product Backlog，通过收集、分析各项业务需求的优先级并确保 Product Backlog 中的 PBI 按照发布计划的时间安排由远到近进行从粗粒度到细粒度的演进。一般会按照优先级制定发布计划，并采用新增、删除、重新估算、提炼（Refine）等几种常见的 Product Backlog 梳理方法。

我们知道在管道中的开发任务分为硬性任务和软性任务两大类，在 Scrum 的 Product Backlog 中同样提供了类似的概念确保 Sprint 执行过程中的弹性化处理。如图 8-8 所示，在一个发布计划中，存在三种类型任务，即必须要有的任务（must have）、最好有的任务（nice to have）和不会有的任务（won't have）。必须要有的任务类似于管道理论中的硬性任务，而最好有的任务相当于软件任务，在一定程度上舍弃部分最好有的任务不会影响到本次发布。

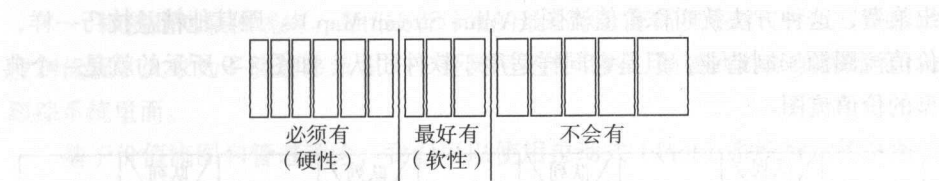


图8-8 Product Backlog中的硬性任务和软性任务

8.2.3 精益与消除浪费

精益（Lean）思想来自制造业，21 世纪初由 Tom 和 Mary Poppendieck 引入软件开发领域^[32]，精益的很多思想被认为是对软件行业有参考价值。与 Scrum 所提供的过程管理框架不同，精益更多体现为一种思维方式，精益的思维方式也常被称为精益思想（Lean Thinking）。

1. 精益思想

精益是敏捷开发的一个重要部分。当把精益制造的一些思想应用到软件开发中时，我们从敏捷开发的其他组成部分中借鉴了一些东西。和敏捷开发一样，精益思想也包含一些价值观^[33]，这些价值观中的一部分如尽快交付、增强学习等与敏捷思想完全一致，但也有一些思想代表着精益的特有思维，最具代表性的就是消除浪费，即找出那些不能直接帮助你创造出有价值软件的工作，然后把它们从

项目当中去掉。关于软件研发过程中的浪费现象可以总结为：

- 部分完成的工作：部分完成但没有最终落地的工作
- 未应用特性：开发完成但没有被客户应用的特性
- 额外过程：开发过程中不需要的流程和中间产物
- 再次学习：人员、环节变动导致反复重新学习
- 信息移交：隐性知识信息的传递总是伴随信息丢失
- 任务切换：多任务工作会导致效率下降
- 资源依赖：因任务或资源相互依赖而导致工作停滞
- 系统缺陷：解决缺陷活动本身就是浪费

对这些浪费现象的分析思想来自于丰田制造系统（TPS）^[14]，并在软件行业中得到映射，精益软件开发过程的倡导者们虽然为我们总结了这些浪费现象，但对如何识别这些浪费进而消除和压缩这些浪费都没有提供很明确的实践方法。我们需要在日常研发过程中观察这些浪费现象进而找到消除和压缩浪费的工作方法。

2. 精益中的管道管理

在《Lean Software Development》^[32]一书中推荐了一种简单的方法来帮助你找出浪费，这种方法就叫作价值流图（Value Stream Map）。跟其他精益技巧一样，价值流图源于制造业，但是它同样适用于软件团队，如图 8-9 所示的就是一个典型的价值流图。

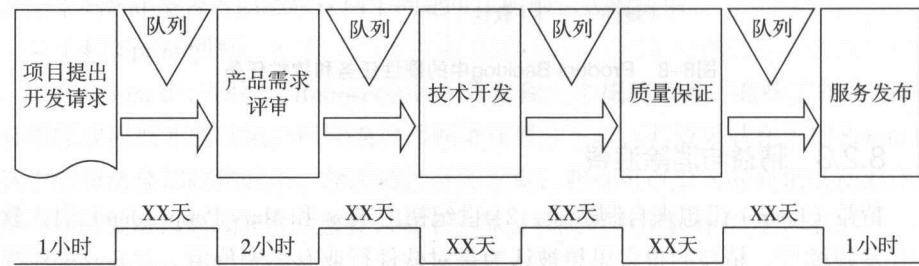


图8-9 价值流图示例

要想得到类似图 8-9 中的价值流图也比较简单，首先从团队已经开发并交付了的一个产品功能开始，回想这个功能从设想到交付经历了哪些步骤。在纸上为每一个步骤画一个方框，用箭头把各个方框连起来。接下来，估计一下完成第一步需要多少时间以及第一步完成后要等多长时间才能开始第二步。对每个步骤进行相同的估计，并且在方框的下方画线来表示工作和等待的时间。

价值流示意图清晰地显示了该流程中涉及多少等待的时间，从团队开始该项目到最终交付，一共花了多少天。在这些天数中，有多少天花在了等待而不是工

作上。这些等待时间可能是由多种原因导致的：需求文档可能需要很长时间才能送交所有的评审者；或者工时估计会议必须延后，因为大家已经有其他安排了等等。价值流示意图展示了各种延迟所造成的累加效果，通过这种整体的影响，可以帮助你进一步探究哪些延迟是浪费，哪些是必要的。

结合管道理论，实际上图 8-9 中的每一个步骤都可以看作是一个管道，所以价值流图就是各个管道的负载流转图。如果下一个管道在等待上一个管道的结果，那么这个管道就处于空置状态，而如果某一个管道上的开发任务与其他管道上的开发任务比例严重不匹配，就会导致因为管道之间数据流转不平衡导致浪费。我们可以经常看到开发过程中出现如下的不良现象：

- 让每个人确认某规格文档要花很长时间，而与此同时开发人员则坐在那干等着项目开工。
- 开发到一半，开发团队意识到软件设计或架构的一个重要部分需要更改，但这会导致严重的问题，因为有很多其他部分依赖它。
- 质量控制团队要等到每一个功能都开发完毕才开始测试软件。
- 分析和设计花费的时间过长，导致进入编码阶段时，每个人都需要加班加点地赶工期。
- 即使是对软件规格、文档或者计划的最小修改都需要经过一个冗长的修改控制流程。大家为了绕过该流程，于是甚至把大型的、颠覆式的修改都放到 bug 跟踪系统里面。

基于价值流图和管道理论，我们可以使用拉动式（Pull）系统帮助团队解决以上问题。所谓拉动式系统，指的是通过使用队列或缓冲区来消除约束的一种运作项目的方法或流程。它也源自日本的汽车制造业^[33]，但对于软件开发也非常有用。与其让用户、经理或者项目负责人把任务、特性、请求“推送”给开发团队，不如让他们把这些请求送入一个队列，由开发团队自己从该队列中拉取。当工作发生堆积并在项目中途导致分配不均衡时，他们可以创建一个缓冲区来解决。开发团队在整个项目中可能会用到好几个不同的队列和缓冲区。事实表明，这是一种减少等待时间、消除浪费的有效方法。

下面我们举一个拉动式系统如何解决问题的例子：软件团队需要等待所有的功能都写入一个大型规格文档，而后者还必须要经过一个冗长的审核流程。或许该流程为的是征求每个人的意见；也许它不过是那些不敢真正承诺的老板或利益干系人的一种保护自己的手段；或者它干脆就是公司一直以来习惯的做事方法，从来没人想到它其实是一种浪费。如果我们用一个拉动式系统来替换它，会是个什么样子呢？

建立一个拉动式系统的第一步是把工作分解成小型的、可供拉取的块。也就

是说，不要编写一个大型的规格文档，而要把系统分解为最小可交付功能，比如一个个独立的用户故事，可能还有针对每个故事的少量文档，这些故事可以单独进行审批。一般来讲，当一个规格文档审查流程长时间停滞不前时，原因通常是大家对某些功能有不同看法。对每个可交付功能进行单独的审批应该至少能够让一部分功能快速得到批准。只要有一个可交付功能通过了批准，开发团队就可以开始进入开发阶段。

拉动式系统可以很好地消除工作的不均衡并防止团队的超负荷工作。上述关于拉动式系统的思想和做法体现的正是 8.1.2 节中介绍的管道负载分析方法，可以认为精益思想也是管理理论的一种具体表现形式。

3. 识别浪费的其他方法

除了价值流图，还有其他一些方法可以用于识别开发过程中的浪费：

（1）项目输入过滤

研发过程通常面向产品，而企业级应用或半互联网半企业级应用的产品最终通过项目实施和落地，这样项目线就成为研发过程的一个重要输入，而项目经理们站在项目线和客户的立场上提出来的需求和计划往往会和产品线、研发线有一定出入。如果本不应该进入研发环节的输入最终进入了研发环节就势必会导致浪费。如何通过规划和分析去把控来自项目上的输入，让项目线需求能够尽量和产品线一致是降低研发成本、消除浪费的一个重要方面，所以项目输入也是我们寻找浪费的一个来源。

（2）会议聚焦

我们不得不经常召开这种或那种会议，那开会是一种浪费吗？很多时候是的。会造成浪费的会议通常会有一些共性，典型的有：输入输出不明确；缺少主持人或主持人不善于引导；会议不是结果导向、无法形成有效决策；会议议程空泛而不能收敛；会议虽然能达成一致，但没有具体工作安排和责任人制度；即使有工作安排但缺乏跟踪和监控机制；会议相关的资料没有充分准备，也没有提前交付到参会人员等。技术管理者需要进行分析 and 识别，看看我们每天的会议中是否具有以上问题。

（3）数据传递有效性对比

目前主流的研发管理方法论中普遍认为沟通和协作是研发成功的关键性因素，而沟通和协作的背后体现的实际上就是数据传递过程的有效性。如果有两个研发团队，其中一个团队中数据从团队中的一个人传递到另一个人的过程无论在时间上或空间上都比另一个团队有效，那两个团队的战斗力无疑是不一样的。数据传递在沟通模式、媒介、工具等各个方面都可能存在效率不高甚至不合理的地方，浪费也就在这些地方不断的滋长，从而消耗着研发团队的战斗力。

（4）管理活动梳理

从管理工作本身入手分析其在工作流程、文档管理、任务分配等各个方面上是否存在冗余或者不合理的管理活动是一种识别浪费的有效实践。管理是需要成本的，管理做得好、做得精细更加需要成本，但管理过程本身也可能像代码一样需要随着研发过程和团队的演变不断进行重构，重构的前提也就是需要我们对管理活动进行分析和梳理，找出其中的浪费之处并进行消除或压缩。

（5）流程执行力

无论是好的管理模式和理念，还是适合团队的研发模型，要想取得令人满意的效果，归根结底还是需要执行力。执行力来自很多方面，如合理的团队组织架构、优秀的人才、高效的工具、良好的团队气氛等，这些通常都不是技术所能起决定作用的领域，却实实在在影响着团队的执行力。流程本身可能是合适的，但因为执行的人不行，或因为工具使用不当导致效率降低，这通常都是属于无法避免但是需要进行压缩的浪费形式。

通过识别，团队中的浪费现象已经摊在大家面前。其中有很大一部分的浪费属于纯粹浪费，这些浪费需要通过一定的思路和工作模式进行消除；而有些浪费通常是必要的，也是不可避免的，对这些浪费而言，我们的思路是尽量进行压缩。关于如何消除纯粹的浪费以及如何压缩必要的浪费可参考笔者的《系统架构设计》^[1]。

8.2.4 看板方法与流程管理

看板方法也是一个制造业的术语，由 David Anderson 引入到软件开发领域。David Anderson 在其著作《看板方法》^[34]一书中这样描述看板方法与精益之间的关系：“看板方法带来了一套复杂的适应性系统，该系统的目的就是在一个组织中催生出精益的效果。”

1. 看板方法

正如 David Anderson 所说，看板方法本身并不是一种软件开发流程或者项目管理方法。使用看板方法之前，你必须已经具备某种流程或方法，而看板方法的作用就在于逐渐改变你已有的流程或方法。事实上，我们也可以把看板方法看作和精益思想一样是一个消除浪费的方法，因为对于软件开发消除浪费就是流程改进的主要目的。从这点上，看板方法和 Scrum 有较大区别，因为 Scrum 主要提供的是过程管理框架，关注于需要做哪些工作，何时做以及谁来做等过程管理类问题。

看板方法为流程管理提供的最基本的工具就是看板（Kanban Board），看板的作用在于把整个开发流程可视化。如图 8-10 所示就是一张典型的看板，看板

上的栏目是：原始需求、计划、开发、测试、产品验收和发布。使用这个看板的团队的工作流程可能是这样的：每个功能都需要经过分析、开发、构建和测试这几个环节。所以，团队可能会从类似图 8-10 那样的一个看板开始，在各栏目中贴上贴纸来代表经过系统中的各个工作项。

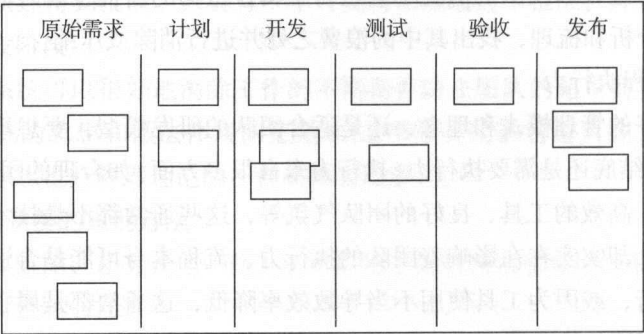


图8-10 正常情况下的看板示例

我们知道 Scrum 中有任务板的概念，看板与任务板表现形式非常相近，但有一个本质性的区别。看板中的每一个栏目中的工作量可以设限制，这点与管道理论有非常密切的关系，我们将在下一节中具体讨论。

2. 看板中的管道管理

我们先来看一个日常开发过程中的典型场景，在某一个特定时期，图 8-10 就是看板的一个快照。如果某一段时间测试人员生病请假，显而易见，看板上在测试这一栏中很快就会堆积很多需要测试的开发任务（见图 8-11）。现在我们有了团队工作流程的一个更加准确的可视化结果，也很容易就发现团队的瓶颈所在。团队中已经有人清楚问题的关键，接下去就是要做开发流程上的改进，为此我们引入一个重要概念，在制品。

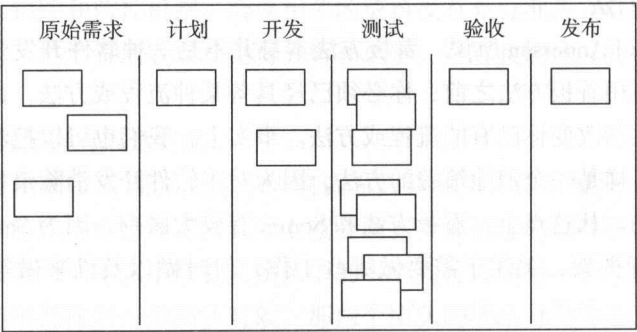


图8-11 测试请假情况下的看板示例

在制品的意思就是当前某个环节中正在开发的任务，而限制在制品就是为这个开发任务数量设定一个上限。在制品是一个看板用语，但我们可以看到它与管道中的开发任务是一致的，而限制在制品就是要管理管道中的负载。管道不能长时间的超负荷运作，看板把工作流程可视化能够帮助团队清楚地看到这种超负荷的问题。在图 8-11 中，我们已经看到测试环节已经处于超负荷运转状态，在这种情况下，马上开始该功能的开发工作就不合时宜了，开发完了之后它也不过是放在那里等着，因为没有多余的人去马上测试它。

当测试工作出现瓶颈，我们有很多选择。给某个环节的在制品设置一个上限意味着限制了可以进入该步骤的任务的数量。这样可以帮助限制团队的选项，从而让团队的选择变得更容易。针对例子中的这个测试环节，如果我们把测试在制品上限设为五（见图 8-12），当第六个开发任务即将进入测试环境时，我们就会发现它已经超过了这个环节的在制品上限。通过在图 8-12 中添加一栏“被测试推迟”的任务，就可以将这个问题充分暴露出来。同时，也就意味着我们需要停下来考虑一下如何解决测试资源不足的问题，解决的一种思路是从其他团队调用一个测试人员过来做短期支援，或者鼓励全民测试，或者其他任何有效的方法。从瓶颈被发现到被解决，整个过程可以在图 8-10、图 8-11、和图 8-12 的流转过程中得到可视化管理。

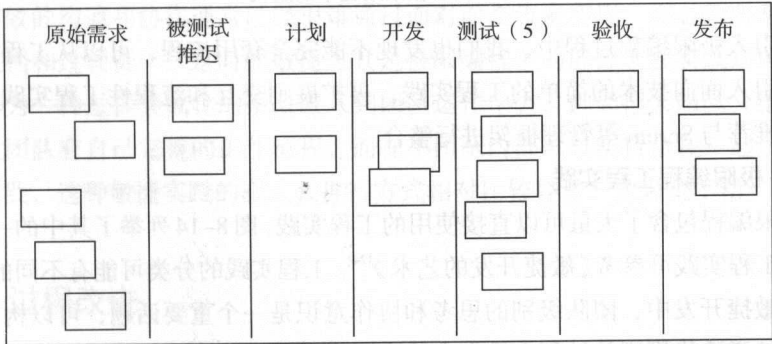


图8-12 测试环节添加在制品上限情况下的看板示例

8.2.5 极限编程与工程实践

前面我们介绍了 Scrum、精益和看板，这三种都偏向于过程管理，代表敏捷方法论的一大派系。在敏捷领域，还存在面向工程实践的另一大派系，代表性的有极限编程（eXtreme Programing, XP）。这两个派系分别从开发方法和管理方法给出了各自关于敏捷方法论的阐释。

针对如何实施敏捷方法，目前主流的做法就是把极限编程与其他过程管理框

架整合起来一起应用，通过 Scrum、看板等先确定软件开发的基本流程和步骤，再通过极限编程中的各项工程实践来具体实现这些流程和步骤。本节对极限编程同样做一个简要介绍。

1. 极限编程方法

极限编程方法试图解决的问题包括软件不能适应需求变化、软件缺陷多、代码质量低、设计不良、项目中浪费大、开发效率低等各个方面。对于如何解决这些问题，极限编程认为可以提炼出一批工程实践用于指导软件开发工作。

回到迭代模型，极限编程是迭代思想的一种表现形式，包括多种迭代形式，分别为分钟级迭代、小时级迭代、每日迭代、数日迭代、一周迭代和季度迭代，每一种迭代形式都有对应的工程实践（见图 8-13）。除此之外的一系列工程实践也确保每种迭代形式能够正确有效的执行。

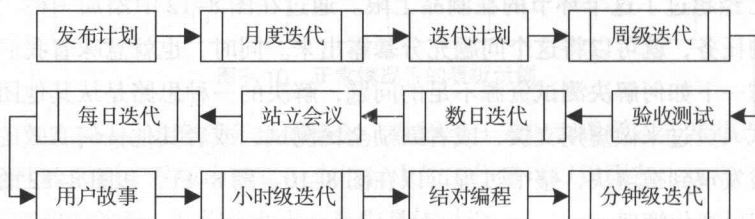


图8-13 极限编程中的迭代

在引入极限编程过程中，我们也发现不能完全套用流程，可以从工程实践出发，先引入面向技术的简单的工程实践，再扩展到交互和流程性工程实践。很多时候也推荐与 Scrum 等管理框架进行整合。

2. 极限编程工程实践

极限编程包含了大量可以直接使用的工程实践，图 8-14 列举了其中的一部分，更多的工程实践可参考《敏捷开发的艺术》^[24]。工程实践的分类可能有不同的标准，通常在敏捷开发中，团队级别的思考和协作意识是一个重要话题，可以构成一个分类；基于迭代模型的计划和发布方法在开发过程中的应用是敏捷区分其他方法论的关键，可以构成一个分类；极限编程从派系上是一种开发方法，开发也可以构成一个分类。

协作与工程实践包括结对编程（Pair Programming）、信息化工作场地、真实用户参与、统一协作语言、站立会议、编码规范、迭代演示、上级汇报等。计划发布与工程实践包括用户故事（User Story）、发布计划、迭代计划、计划游戏和持续集成等。而测试驱动开发（Test Driven Development, TDD）、客户测试、重构、简单设计、试验方案（Spike Solution）等工程实践属于开发范畴。

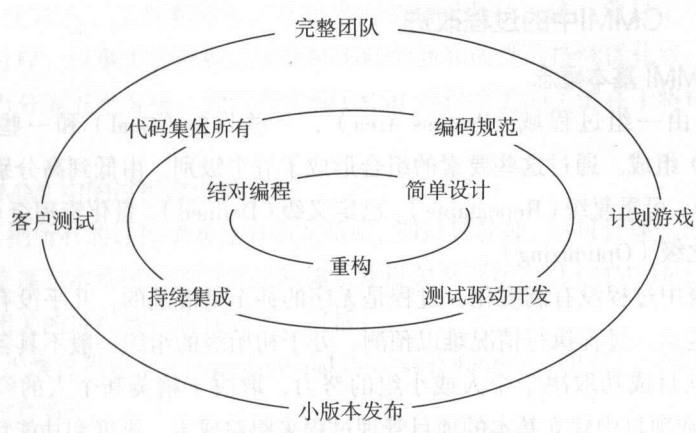


图8-14 极限编程中的工程实践

在诸多极限编程工程实践中，一般认为结对编程和测试驱动开发属于比较难以做到的实践，而信息化工作场地、站立会议、重构、试验方案实施起来相对就容易很多。有些工程实践我们认为推行起来有难度但还是需要我们去实施，包括统一协作语言、编码规范、持续集成、发布计划和迭代计划、故事和估算、客户测试等。

当你的团队规模逐步变大，就需要考虑团队成员之间的有效沟通和协作问题，这种有效的沟通和协作通常已经很难通过面对面交流来实现，而必须要有相对完备的文档和过程资产，这时候敏捷尤其是极限编程中的部分工程实践是可以也是应该作为一种过程集成化的手段嵌入到团队运作中去，例如把大团队分成小团队，大团队有自己完整的工作流程，而在小团队内部可以实行坐在一起、站立会议等实践，这种敏捷实践的嵌入式推行方式相对比较容易，效果也不错。

8.3 过程改进

无论是经典的软件过程模型，或是多种基于管道理论的敏捷方法，都是通过软件过程模型来组织软件开发过程，软件过程模型偏重于开发流程管理，这一节我们通过过程要素的另一个视图来剖析软件开发过程。任何形式的软件开发过程都可以使用流程、人员和技术三个方面来切入。流程定义一系列开发的步骤和操作方法，人员需要通过培训和管理提升技能，而广义上的技术包括应用领域、工具、语言、信息和环境。

业界对如何进行过程改进也存在一些方法论和框架，最具代表性的就是CMMI（Capability Maturity Model Integration，软件能力成熟度模型集成）。而敏捷方法中也提供了许多专门针对过程改进的实践、流程和工具。

8.3.1 CMMI中的过程改进

1. CMMI 基本概念

CMMI 由一组过程域（Process Area）、一些目标（Goal）和一些工程实践（Practice）组成。通过这些要素的组合形成了五个级别，由低到高分别称为初始级（Initial）、可重复级（Repeatable）、已定义级（Defined）、量化管理级（Managed）和持续优化级（Optimizing）。

初始级中过程没有制度化，过程是无序的甚至是混乱的，几乎没有什么过程经过妥善定义，过程执行情况难以预测。处于初始级的组织一般不具备稳定的开发环境，项目成功取决于个人或小组的努力，取决于精英和个人的经验（见图 8-15a）。在项目中建立基本的项目管理过程来跟踪成本、进度和功能特性，制定必要的过程纪律，能重复早先类似项目取得的成功，具备这些特征的就是可重复级（见图 8-15b）。已定义级中，已将管理和工程两方面的过程文档化和标准化，并形成了组织级的过程资产。所有项目都使用经批准和剪裁的标准过程来开发和维护，需要收集数据，也要使用数据（见图 8-15c）。而量化管理级则使用统计和其他量化技术对项目过程进行控制，建立了质量和过程性能的定量目标，作为过程管理的准则，质量和过程性能度量数据能用于支持决策（见图 8-15d）。最终的持续优化级基于对过程中性能偏差的原因的定量分析，通过渐进的和革新的技术改进，持续地进行过程性能改进。组织过程改进得到识别、评估和实施，并且全体员工参与过程优化（见图 8-15e）。

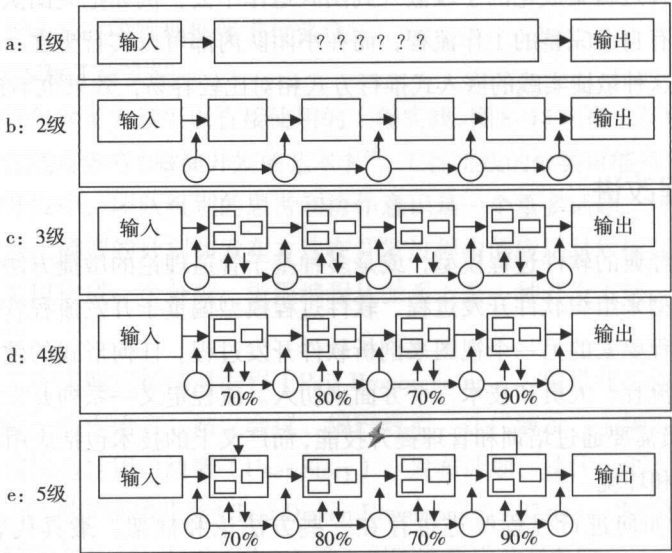


图8-15 CMMI五级模型

CMMI 关注人、工具和方法，从无序的初始级开始，建立项目记录、建立稳定一致的过程、以事实为依据达到能够不断创新和改进的持续优化级，将企业过程成熟能力分为五个等级。我们在实施 CMMI 过程改进的关键在于将标准开发过程制度化。

2. CMMI 中的过程改进核心域

CMMI 把所有的过程域划分为四大类型，即过程管理、项目管理、工程和支持，其中过程管理与本章讨论的研发过程体系建设关系密切。以 CMMI Dev 1.3 为例，过程管理相关的过程域一共有五个，包括：

- 组织级过程定义（Organizational Process Definition, OPD），建立和维护有用的组织过程资产
- 组织级过程焦点（Organizational Process Focus, OPF）：在理解现有过程强项和弱项的基础上计划和实施组织过程改善
- 组织培训管理（Organizational Training, OT）：增加组织各级人员的技能和知识，使他们能有效地执行他们的任务
- 组织过程性能（Organizational Process Preformance, OPP）：建立与维护组织过程性能的量化标准，以便使用量化方式管理项目
- 组织性能管理（Organizational Performance Management, OPM）：用量化的目标来驱动，并且用量化的方式来监控改进的效果

CMMI 又根据目标等级的不同把这五大过程管理类过程域分成基本过程管理类过程域和高级过程管理类过程域两大类。图 8-16 和图 8-17 描述了两类过程管理类过程域与其他过程域类别之间的互动关系。关于 CMMI 的详细讨论不是本书的重点，更多内容可参考相关资料^[35]。

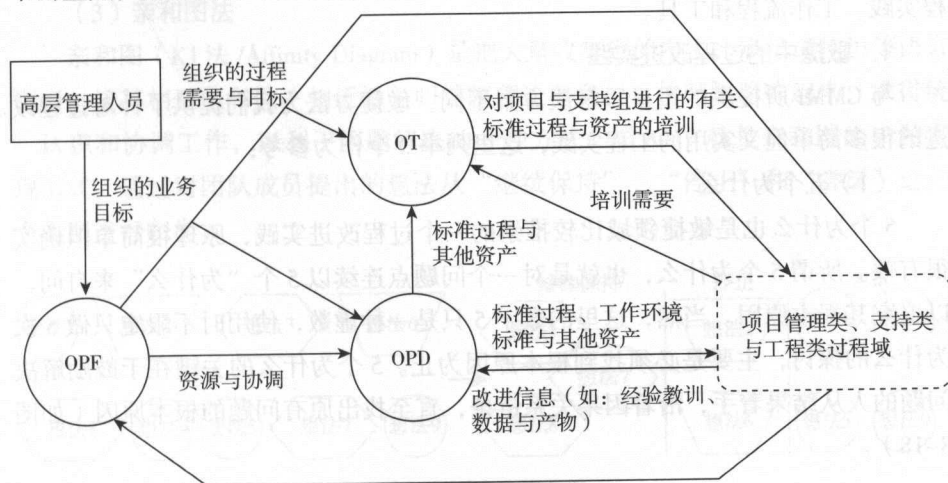


图8-16 基础过程域之间以及与其他过程域类别之间的互动关系（摘自CMMI Dev 1.3）

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

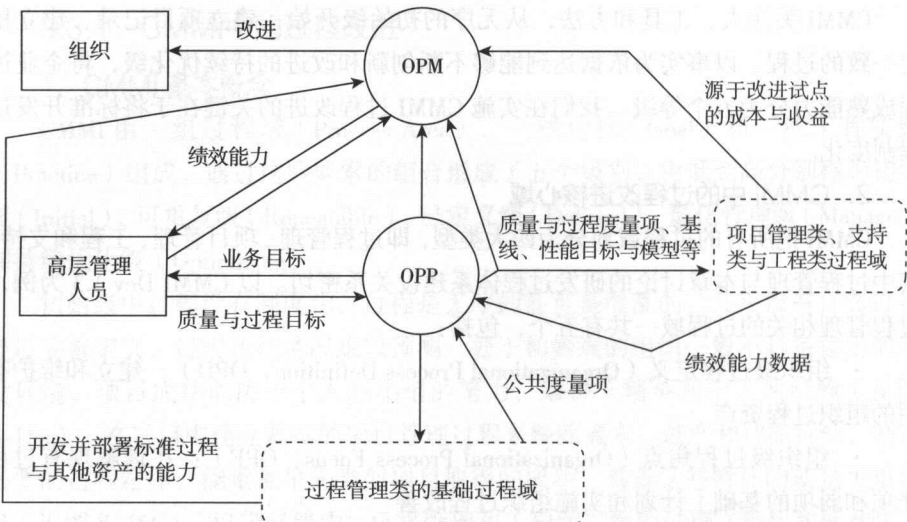


图8-17 高级过程域之间以及与其他过程域类别之间的互动关系（摘自CMMI Dev 1.3）

8.3.2 敏捷中的过程改进

敏捷思想中有一条原则指导我们进行过程改进：每隔一定时间，团队会针对如何才能更有效地工作方面进行反省，然后相应地对自己的行为进行调整。很多不确定性因素会导致计划失效，比如项目成员增减、技术应用效果、用户需求的改变、竞争者对我们的影响等都会让我们做出不同的反应。敏捷不是基于预定义的工作方式，而是基于经验性（Empirical）的方式，对以上这些变化，敏捷团队通过不断的反省调整来保持团队的敏捷性。敏捷为我们提供了过程改进相关的工程实践、工作流程和工具。

1. 敏捷中的过程改进实践

与 CMMI 所提供的重量级解决方案不同，敏捷方法为我们提供了针对过程改进的很多简单但又实用的工程实践，这里列举 3 个作为参考：

（1）五个为什么

5 个为什么也是敏捷领域比较推崇的一个过程改进实践，原理很简单但确实很有效。所谓 5 个为什么，也就是对一个问题点连续以 5 个“为什么”来自问，以追究其根本原因。当然，这里的数字 5 只是一种虚数，使用时不限定只做 5 次为什么的探讨，主要是必须找到根本原因为止。5 个为什么的关键在于鼓励解决问题的人从结果着手，沿着因果关系链条，直至找出原有问题的根本原因（如图 8-18）。

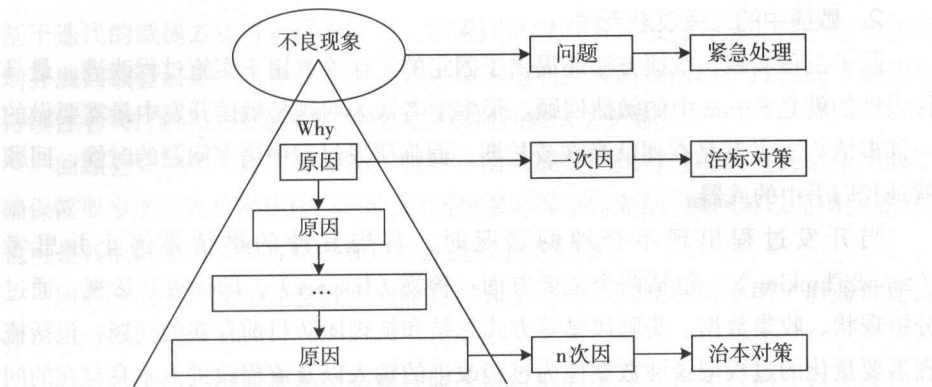


图8-18 5个为什么方法

(2) 头脑风暴

头脑风暴（Brain Storming）是我们经常使用的一种触发灵感的活动，下面是需求和 UI 界面匹配问题团队成员之间一次简单的头脑风暴，通过讨论大家就工作流程做了一项改进：

开发人员：现在由于开发周期比较短，产品经理在没有完全梳理清楚产品需求的情况下就召开需求评审会议，导致后期开发过程中出现很多问题，返工的现象比较严重。
技术管理者：我们可以开展二次评审，也就是说在面向所有开发人员的正式需求评审之前，先可以找部分技术代表作第一次评审，通过第一次评审对需求存在的问题先修订之后再召集所有人开第二次评审
产品经理：那我在产品初稿出来之后会找前后端的技术负责人先碰下
项目经理：恩，下一个版本我们在流程上就引入二次评审

(3) 亲和图法

亲和图（KJ 法 /Affinity Diagram）是把大量收集到的事实、意见或构思等语言资料，按其相互亲和性（相近性）归纳整理这些资料，使问题明确起来，求得统一认识和协调工作，以利于问题解决的一种方法。图 8-19 就是亲和图的一种表现形式，通过对团队成员提出的想法从“继续保持”、“停止”和“尝试”这三个类型进行归类。

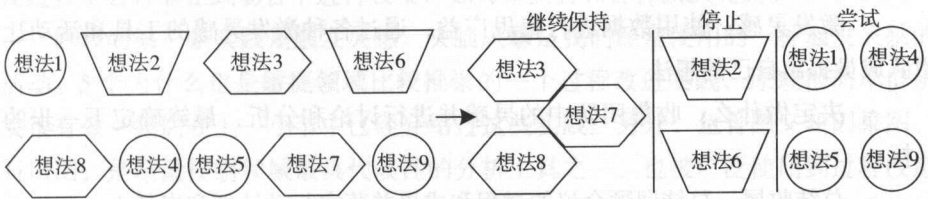


图8-19 亲和图示例

2. 敏捷中的过程改进流程

除了工程实践，敏捷方法还提供了固定的工作流程用于实施过程改进，最具代表性的就是 Scrum 中的敏捷回顾。很多学者认为回顾是敏捷开发中最需要做的一件事情^[13]，尤其是在团队尚在成长期，面临研发过程中诸多问题的时候，回顾就是我们手中的武器。

当开发过程出现不合理的状况时，首当其冲的事情是停止并思考（Stop&Thinking），包括两个主要方面：检视（Inspect），即检查和诊视，通过分析现状、收集数据、头脑风暴等方式总结和梳理团队目前存在的问题，包括梳理需要量化的过程记录性数据作为过程改进的输入以及流程改进点本身存在的问题这两个主要方面；调整（Adapt），即过程改进，有了量化的数据和流程改进的突破点，就可以确定调整策略和下一步的工作计划，并落实责任人。

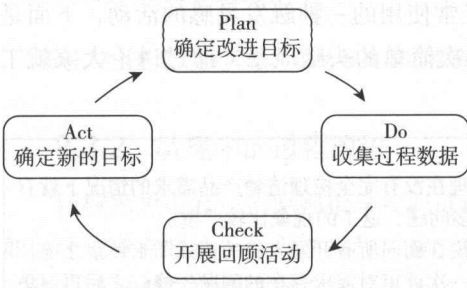


图8-20 PDCA环与回顾

Stop&Thinking 体现的就是回顾思想，回顾是过程改进中诸多方法论和实践的一个集合体，也和质量领域中的主流思想保持一致，回顾本身也是图 8-20 中 PDCA 环的一种具体体现。

（1）回顾会议

敏捷回顾的开展方式主要是通过回顾会议。开会的目的在于分析数据、剖析流程并在团队级别形成统一认识。回顾会议的流程可以分为以下 5 个步骤，其中前两步需要在会议前完成。回顾会议的目的是对研发过程中的客观数据进行分析从而得出结论和行动计划，当然各个团队可以根据实际情况进行调整和裁剪，但会议的输入、输出和议程应该是一致的：

- 确定目标：确定本次回顾会议中需要改进的目标，一般一次回顾 1~2 个目标比较合理
 - 收集数据：为了对改进过程有量化的标准，需要进行数据收集。数据来自日常研发过程中与该改进目标相关的方方面面
 - 激发灵感：使用数据进行集思广益，通过各种激发灵感的工具和活动让团队成员提出自己的想法
 - 决定做什么：收集团队中的灵感并进行讨论和分析，最终确定下一步的目标
 - 总结收尾：总结回顾会议的过程和成果并落实行动计划和责任人
- 在 Scrum 中，回顾会议召开的时间是在迭代结束之后，前提是团队正在使用

基于迭代的敏捷方法进行研发管理。如果团队没有采用敏捷方法，以一定节奏定期开展回顾会议是一项最佳实践，如果是不定期开展回顾活动，过程的节奏感和持续性会大打折扣，可能导致后续大家都对回顾失去兴趣。

回顾会议的参与者不要太多，包括：团队成员，执行过程改进的整个团队，确保需要参加的人员都到位；顾问，顾问很多时候是需要的，顾问可以是团队成员，也可是其他研发团队中有回顾和过程改进经验的同事。

回顾会议的持续时间会因为回顾目标和主题的不同而有所不同，但通常建议在半小时到一小时之间。

回顾会议的产出通常是团队对具体改进目标的统一认识和行动计划，这些行动计划有些面向具体一个工作点，那么改进的效果相对比较容易衡量；有些则面向工作流程，而流程的改变通常需要一段时间，故这类行动计划的有效性需要在后续的回顾会议中时常进行总结和讨论以确定其在推进的方向上是否还具有时效性和可行性。

（2）回顾的工具和实践

本节将根据上文中的回顾会议流程进行一些场景分析，帮助读者了解和掌握回顾会议中的具体做法和注意点。文中的实例可能并不适合所有的团队，仅供参考。

回顾的第一条实践是确定目标，可以通过两个活动来确定目标，分别是“聚焦”和“不良事件响应”。聚焦是主动地去关注研发过程中的某一个潜在需要改进的点，而不良事件响应通常是因为发生某个事故导致我们不得不动地去应对这些事故并探讨是否有可以避免该类事故再次发生的方法。从质量成本的角度讲，聚焦是一种一致性成本而不良事件响应是不一致性成本，日常研发过程中应多采用聚焦的方法以降低质量成本。

回顾的第二条实践是收集数据。收集数据一个有效的活动就是“时间表”，如果团队维护着这么一份时间表，则每天发生的问题都会有一个明确的记录，这些记录就为我们的回顾活动提交了无可辩驳的回顾数据，通过对这些数据进行分析，我们就能进行灵感的触发，从而找出解决问题的思路。数据也可以在日常研发过程中各种非正式场合下进行收集，如每日例会和各种团队交流。

回顾的第三条实践是触发灵感。头脑风暴是我们经常使用的一种触发灵感的活动。5个为什么也是敏捷领域比较推崇的一个过程改进实践，原理很简单但确实很有效，我们在上一小节中已经介绍过这些实践。另外，鱼骨图又叫因果图、石川图，是质量控制领域最具代表性的分析工具之一，也被广泛使用到过程改进中。可以从研发团队的各个方面出发找出导致开发效率低的原因，从而为我们后续的行动计划指明方向。

关于决定做什么，虽然后续工作安排取决于具体的目标，但有两个方面我们需要注意。其一是关注流程，从点覆盖到面是我们梳理流程的一个目标。例如，从 Jenkins 打包失败这个点提出在流程上需要进行提交代码前先本地打包这一覆盖到面的改进；再如，从测试环境部署失败看流程问题的话就需要我们先搭建开发环境然后在发布前先部署开发环境。其二在于使用 SMART 原则，SMART 原则是指具体（Specific）、可衡量（Measurable）、可实现（Attainable）、相关性（Relevant）和有时间限制（Time-bound），是目标管理领域的一个核心原则。我们在制定行动计划时如何判断改进工作安排是否合理和可行很大程度上可以参考这一原则。

当回顾会议接近尾声时，我们需要做会议总结，可以采用“+/ Δ ”方法。通过分析目前做得比较好的工作（+，代表需要进一步加强）和尚待改进的工作（ Δ ，代表需要进一步改进），团队成员在意识形态上达成一致有助于改进工作的顺利开展。

3. 敏捷中的过程改进工具

敏捷的各种方法中存在一些工具可以用于帮忙我们实施过程改进：

(1) 燃尽图和燃烧图

燃尽图（Burndown Chart）和燃烧图（Burnup Chart）来自于 Scrum，用于作为专门的工具对 Sprint 执行过程进行监控。燃尽图（见图 8-21a）关注于剩余功能，而燃烧图（见图 8-21b）关注已完成功能。我们在燃烧图中添加了一条异常情况下的曲线（图 8-21b 中位于下方的曲线），可以看到这种异常很大程度上源于使用了 Water-Scrum-Fall 模式，在 Sprint 的前一半时间内，团队并没有产出任何可交付成果导致后半程发力不足。

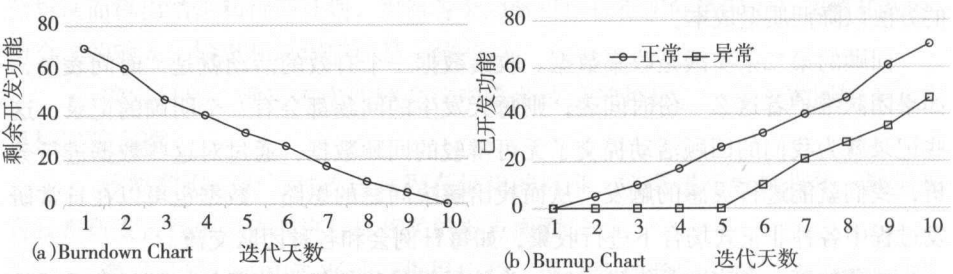


图8-21 Burndown&Burnup Chart

(2) 价值流图

价值流图来源于 TPS^[33]，TPS 的核心思想是消除浪费。价值流图的作用就是通过可视化的效果让我们能够发现这些问题，从而通过拉动式系统等方式解决问题以达到持续改进的目的。关于价值流图的详细讨论可以参考前文 8.2.3 节

中的内容。

(3) 累积流量图

管道理论的一大话题是我们需要测量并管理管道中的流量，而在看板方法中，累积流量图（Cumulative Flow Diagram, CFD）为我们提供了这方面支持。图 8-22 就是一张累计流图示例，可以看到测试所对应的条带在变宽，这就告诉我们测试可能正在成为整个工作流中的一项瓶颈。图中还可以添加一些额外的线，用来表示平均到达速度（Arrival Rate，每天新增的工作项的数目）和平均工作存量（Inventory，工作流中工作项的总数），并展示平均交付时间（Lead Time，每个工作项在系统中存在的时间）。

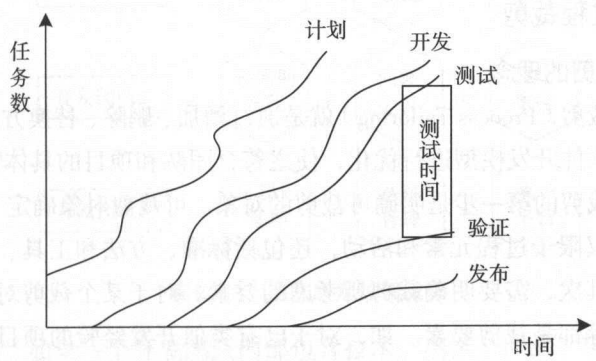


图8-22 累积流图示例

使用 CFD 管理系统流量的关键在于寻找那些能够表明存在问题的特征。看板可以告诉你今天你的工作流中哪里存在不均衡，并且帮助你通过添加在制品上限的方法来管理每一天的工作流。而 CFD 的作用则是让你能够看到“在一个时间段内，你的整个流程的表现如何”，这样你就可以采取措施来找出并修复那些长期的问题。

结合上述限制在制品上限的方法，就可以通过 CFD 试验在制品上限的值并管理系统流量。一个软件开发的过程一开始可能处于不稳定的状态，通过对开发、测试等各个环节设置不同的在制品上限值，并观察对应的平均交付时间，找到基于现有资源的最合适的在制品上限值组合。

8.4 建立合适的过程体系

本章前面几节介绍的过程体系都有其独到之处，但也各有其局限性，如 PACE 和 IPD 主要面向大型团队且需要与团队绩效等组织层面工作模式紧密结合，中小型团队通常不适合；CMMI 模型只是回答了“达到 XX 级别的软件过程应该

长什么样”这个问题，并没有说明“怎么做才能达到 XX 级别”，而且本身也是过于复杂，中小型团队实施过程需要精简；SCRUM 模型看似简单，但对团队要求太高，要想达到理想效果难度偏大；XP 偏重于工程实践，在管理理论体系上不够完善；LEAN 目前还处于理论体系研究和完善阶段，缺乏具体可操作的模式和工具；PMBOK 则偏重于通用性的项目管理，而不完全适合软件开发过程的管理。很多团队尤其是初创型团队或亟须过程改进的团队通常不建议也无法照搬上述模型中的任何一个，而需要结合团队实际情况博采众长和集思广益，这就需要对上述过程管理系统进行裁剪。

8.4.1 过程裁剪

1. 过程裁剪的理念

所谓过程裁剪（Process Tailoring）就是通过增加、删除、替换方法、修改顺序、组合等方式对软件开发模型进行优化，使之符合团队和项目的具体特点。

进行过程裁剪的第一步是明确可裁剪的对象。可裁剪对象确定了裁剪的范围，可裁剪对象不仅限于过程元素和活动，还包括标准、方法和工具、输出的工作产品及模板等。其次，需要明确裁剪所考虑的要素。对于某个裁剪对象，其范围、频度、正式度等都是裁剪要素。如，对于已有类似开发经验的项目，可以适当减少过程培训、业务培训等活动；对于开发周期较短的项目，可以适当合并一些评审活动，如概要设计和详细设计评审合并进行。当我们对活动、文档、度量指标等进行裁剪并形成新的过程体系之后，一般也需要评审该过程体系并把它上升到基线标准。

采用简单易用的图表等形式展示裁剪的指南有助于裁剪工作的推进，对上述提到的裁剪对象、裁剪的方法应都有描述，可以针对不同类型的项目或不同类型的活动提供裁剪后的几套模板，并确保裁剪指南的描述没有二义性，确保减少沟通的误差。

2. 过程裁剪的几种表现形式

通过过程裁剪我们形成了一定的过程体系，这些裁剪活动也具有层次化特征，即不同级别所应用的裁剪方法和结果可以是不一样的。我们分别从应用级别、团队级别和行业级别描述过程裁剪的几种表现形式。

（1）应用级别

顾名思义，应用级别的过程裁剪指的是每个应用对裁剪的标准和原则是不一样的，经裁剪所获得的过程体系自然也不一样。结合各个应用的自身特点灵活采用软件开发过程很大程度上满足敏捷的价值观。

有些应用的开发可能偏向纯敏捷型，而有些则可能带有传统瀑布模式下的工

作方式。对前者而言，短周期迭代是必须要采用的一种工程实践，但对于后者而言，可能很难具备固定的迭代周期，则可以从每日例会、回顾会议、持续集成等与生命周期关系并不是很紧密的工程实践入手。

(2) 团队级别

图 8-23 展示的是另一种过程裁剪的维度，即团队级别。开发团队根据是否专职于某一项职能、是否分布在不同的地理位置、是否有专职的敏捷管理人员等可以分成不同的类型，也可以采取不同的开发流程和工程实践。

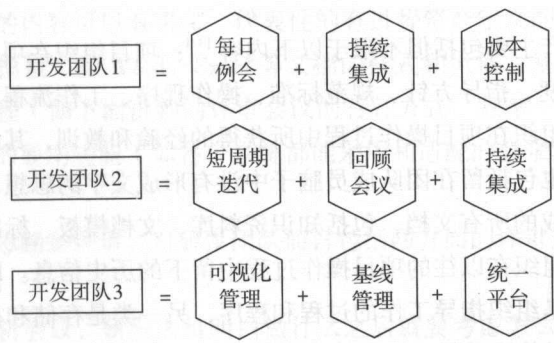


图8-23 团队级别过程裁剪

举例来说，如果一个开发团队内部包含技术、测试、产品、项目等多个角色，那么推行 Scrum 这样的全生命周期的过程管理框架相对会比较容易一些；反之，由于不同职能团队之间需要较多的跨团队协作，优先使用持续集成、回顾会议和可视化管理可能是不错的选择。

(3) 行业级别

行业特性有时候也能影响到过程裁剪。医疗行业的最大特定就是作为用户的患者数据和作为客户的医护数据都位于医院内部，外部系统无论是传统的 HIS、LIS、PACS 系统还是现在流行的各种互联网化 APP，都需要与医院的信息系统进行集成，这样类似配置管理、持续集成等工程实践就需要发挥其作用。同时，由于很多应用是面向医院，所以为了把控产品需求的正确性和实效性，也建议与客户坐在一起、使用统一语言等敏捷工程实践。另一方面，由于与医院的合作通常会基于项目实施的方式开展工作，因此传统的项目管理的过程和工具同样在医疗行业中得到普遍应用。

8.4.2 过程资产建设

组织过程资产是项目管理体系的一个核心概念，而作为技术管理者，同样需要从技术角度出发进行过程资产建设，通过梳理、设计组织过程资产在实际研发

过程中的分类标准以及各类组织过程资产的来源渠道，使得开发团队抽象的组织过程资产概念有了相对清晰、具象的认识，能够在一定程度上提升组织中关于组织过程资产沉淀的理解及执行力度，为团队的持续学习与发展奠定基础。

1. 过程资产的概念

所谓组织过程资产（Organizational Process Assets）指一个学习型组织在项目操作过程中所积累的无形资产。组织过程资产的累积程度是衡量一个项目组织管理体系成熟度的重要指标，项目组织在实践中形成自己独特的过程资产，构成组织的核心竞争力。

组织过程资产主要包括但不限于以下内容^[37]：项目组织在项目管理过程中指定的各种规章制度、指导方针、规范标准、操作程序、工作流程、行为准则和工具方法等；项目组织在项目操作过程中所获得的经验和教训，其中既包括已经形成文字的档案，也包括留在团队成员脑子中没有形成文字的思想；项目组织在项目管理过程中形成的所有文档，包括知识资料库、文档模板、标准化的表格、风险清单等；项目组织在以往的项目操作过程中留下的历史信息。以上内容可以分成两大类，一类是组织指导工作的过程和程序，另一类是存储和检索信息的组织公用知识库。前者关注过程，后者关注数据。

组织过程资产维护方式主要有两个方面：一是按组织过程资产分类标准在项目管理系统中的项目资产库中予以展现；二是将维护组织过程资产作为项目管理工作的重要环节，项目负责人按不同的项目工作阶段对信息收集整理，并在项目资产库中进行维护。这些方式对于研发过程资产建设中也同样适用，但由于技术研发的自身特点，研发过程资产的建设也应该有其特定的思路和操作方式。

2. 研发过程资产建设的思路

如果研发团队中出现以下症状或表象就说明研发过程资产建设是欠缺的：没有文档记录导致过程资产丢失；开发人员个人素质要求偏高；交接困难，新员工难以熟悉现有模块的设计思路；缺少设计评审，设计问题延后暴露。面对以上问题，过程资产建设的思路主要有两个方面。

（1）研发知识管理

研发知识管理是一个组织级别的活动，重点关注：

- 知识服务于业务

从具体业务出发、而不是从技术出发管理研发知识。技术人员进行知识梳理的过程中（尤其是刚开始的阶段）往往习惯于从技术本身出发来看问题，导致梳理出来的东西只有技术团队内部有限的几个人能看懂。业务领域模型才是一个系统的核心，技术只是这一模型的一种实现方式，技术人员梳理的知识同样也要让

产品线、项目线能够参与讨论和总结。

· 系统运作与界限

关注系统的运作流程及其服务提供的界限。对多团队协作的研发过程而言，系统集成是团队之间协作的根本任务，如何定义系统之间的服务边界，并把这些边界梳理成统一接口是团队作为服务提供者对外所需要暴露的知识；对内而言，各个服务的逻辑流程是确保团队内部高效运作的要求。

· 通用库

通用库包含的内容可以有多种，代表性的有过程资产定义的格式和团队交流采用的信息传递模板（如 Word、Excel 等文档的样式和风格、基本章节的划分和定义）、工作流程（如下面讲到的评审会议的召开方式、频率）以及在团队和组织级别进行提取的通用功能（如各个系统都能采用和适配的登录注册功能）。

（2）评审

另一个主要思路是评审。对研发团队而言包括两方面的评审过程：

· 设计评审

通过需求分析会议，研发人员明白做什么之后就要考虑怎么去做，这个阶段就是要进行系统设计。系统设计的开展方式上可能由相对资深的研发主管主导，也有可能是团队中的任何负责某块功能的开发人员。无论哪种方式，当系统设计的成果出来之后、正式启动代码开发之前，设计评审是必须要有的，以确保设计中没有业务逻辑纰漏以及在开发团队成员之间达成一致。

· 代码评审

当一个阶段的代码已经基本开发完成并提交测试，是时候大家一起坐下来进行代码评审。代码评审通常的方式是同级评审（Peer Review），包括代码走查、代码审查等形式并灵活运用各种重构手段确保代码的质量。通过加强过程资产建设和评审，轻量级研发知识管理的目标在于形成“老人做新产品，新人做老产品”的研发良性循环。

3. 研发过程资产建设的工具与实践

作为技术管理者，当同样的步骤需要重复发生、当信息传递因为人而中断、当基础信息需要不断保持最新、当有人事变动需要交接时发现很多平时都在用的研发相关信息一下子都变得找不到了，这时候需要意识到缺少一个平台，一个团队知识管理的平台。

（1）工具

团队知识管理有很多潜在的需求，如我们需要在统一的空间中管理信息资源，包括文本信息、表格数据等；图片信息、手写信息、各种图形符号；来自网络当中的各种信息；甚至要涉及音频、视频、影像资料。同时，能够随时随地查找所

需信息，团队之间的高效协作与信息共享；并确保信息的安全性和可靠性。通过知识管理平台的建设我们的目的就是要满足这些需求，这些需求可以理解为过程资产建设在研发团队中的具体体现。

团队知识管理平台在数据管理上应该具有一些通用功能。首先数据应该在公司内部，目前市面上有很多托管的数据管理平台，但这些平台中的内容基本都是维护在公网环境中，相比数据维护权限掌握在别人手里，把数据保存在公司内部显然从安全性和操作性上都更优；方便团队协作，解决上述问题的初衷就是更加有效地进行团队协作，本节也是从团队知识管理而不是个人知识管理的角度出发进行探讨；内外网联通，数据保存在公司内部，也就是内网环境的话，需要进行内外网互通确保在公司外部环境下通过 VPN 等访问方式同样可以使用知识管理平台。

市面上支持团队知识管理需求的工具有很多，例如最简单的就是使用 Office 自带的 OneNote，也可以自己搭建各种 wiki 系统，另外像 Confluence 这种专业的企业知识管理与协同软件也是很不错的原则。但工具总归只是工具，关键在于使用工具的方法，即流程和工程实践。

(2) 流程

研发知识管理包括设计会议流程和代码质量保证流程，流程中的知识表现形式将在实践部分进行介绍。设计会议的通用流程包括根据需求进行业务抽象→建立系统模型→确定系统运作流程和服务边界→评审与存档等步骤，其中根本的要求就是要进行系统建模和确定服务边界。而代码质量保证的通用流程如下，其中重点把握代码评审部分（见图 8-24）。

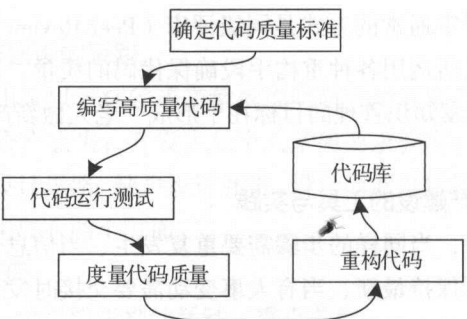


图8-24 代码评审流程

(3) 工程实践

工程实践的产出是代表着知识管理思想的具体产物，包括：

- 模块责任制

模块责任制的初衷是确保知识管理过程的完整性，无论领域模型、业务流程

还是其他多个知识方面都通过业务模块这一基本单元进行组织。模块责任人负责整个模块的知识管理，在具体文档的组织上也建议一个模块维护一份文档。模块责任制的示意图参考图 8-25。

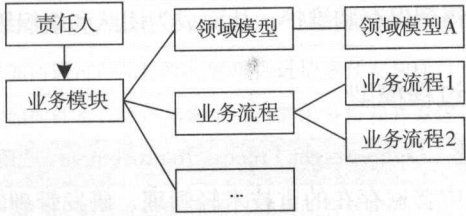


图8-25 模块责任制示意图

· 领域模型

按照领域驱动的设计思想^[7]，一个领域模型中应该包括上下文、实体、事件、存储库等内容，其中最常用且最必要的就是上下文领域模型，图 8-26 是一个典型的上下文领域模型示例图，来自《系统架构设计》中的案例^[1]。

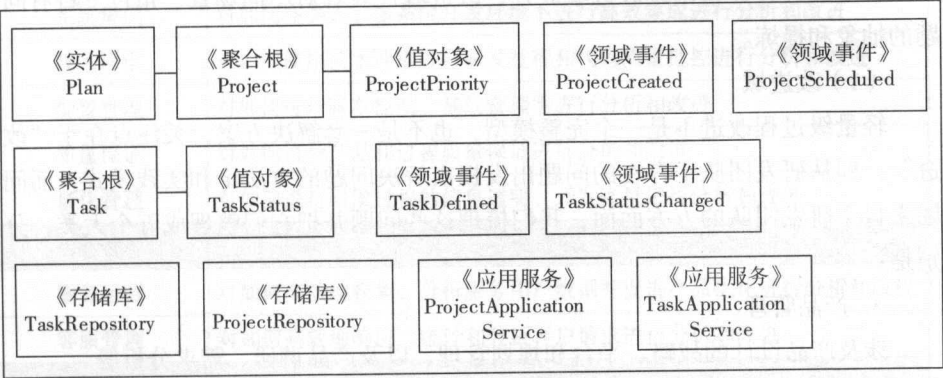


图8-26 上下文领域模型示意图

· UML 建模

统一建模语言是用来对软件密集系统进行可视化建模的一种语言，常用的用例图、时序图、状态图和活动图等都一定程度同时面向业务和技术，其中时序图主要展示业务间的详细流程，同时显示了流程中不同对象间的调用关系和调用顺序。结合把控“系统运行和边界”的知识管理思想，时序图能够详细描述前端调用接口，定义接口名称、调用方式及输入输出条件；同时也能描述后端各组件之间的调用关系及调用顺序，直观展现了业务模型，是 UML 中最适合做研发知识管理的一个必备工具。

· 团队代码评审

Code Review 很多团队都在进行，结合“模块责任制”实践，我们可以以模块

为 Review 的基本单元，采用定期（建议一周一次）的集体式（包括团队所有开发人员）的方式进行评审。

以上流程和工程实践都很容易实行和推广，所产生的中间以及最终产物都应该在知识管理平台中得到保存和维护，从而成为团队和组织级别的过程资产。

8.4.3 轻量级过程模型

轻量级过程改进（Light-weight Process Improvement, LPI）是一种针对中小型团队软件研发过程中普遍存在的重技术轻管理、研发管理缺乏规范、过程改进理念淡薄等现状和问题而整理的一种软件过程改进方法和规范，有众多轻量级过程改进域组成，主要对中小型团队持续地改进其软件过程能力提供一些参考，很多改进域可能只局限于特定团队和场景，需要大家根据各自团队的现状做裁剪和扩充。

1. 轻量级过程建模

轻量级过程建模围绕改进域展开，改进域代表着对不同场景、角色下特有问题的抽象和提炼。

（1）改进域

轻量级过程改进不是一个完整模型，也不是一套解决方案，关注点在于“改进”，即从研发团队中存在的问题出发提出解决问题的方法论和实践模式，而问题来自于研发团队的方方面面，我们梳理这些问题并把它们整理成五个大类，分别是：

- 产品管理

涉及产品设计的战略、平台和规划管理，以及产品调研、需求分析等

- 项目管理

涉及从项目启动到项目验收的全过程，主要把握范围、计划等维度

- 研发管理

涉及从需求到服务交付的全过程，主要把握系统设计、实现等维度

- 运营管理

涉及如何进行服务发布、市场推广、客户请求管理等方面

- 团队管理

涉及团队的组建、培训、协作以及过程资产建设等方面

上述五大类涵盖了一个研发团队中需要进行过程改进的主要工作领域，每个大类中我们再根据工作的性质和内容细分具体的改进域。例如表 8-1 就是本书梳理 20 个改进域，供读者参考。

表8-1 研发团队改进域列表

产品管理	改进域描述
高效决策	对如何进行有效决策以促进团队对产品开发达成一致进行分析和改进
产品平台	对如何创建产品化平台以促进产品的稳定构建进行分析和改进
标准化管理	对如何通过产品标准化管理促进快速开发和项目实施进行分析和改进
需求开发	对如何针对具体产品需求进行需求分析和开发进行分析和改进
项目管理	改进域描述
项目启动	对如何从销售线过渡到项目线从而有效启动项目进行分析和改进
项目计划	对如何创建、管理项目范围和时间进行分析和改进
需求管理	对如何进行需求调研、需求管理进行分析和改进
项目监控	对如何进行需求变更、问题跟踪和风险管理进行分析和改进
客户验收	对如何进行产品试运行和验收进行分析和改进
研发管理	改进域描述
系统集成	对如何在多人、多系统开发环境下进行高效集成进行分析和改进
技术评审	对如何使用技术评审进行研发过程和代码质量把控进行分析和改进
配置管理	对如何进行版本控制、基线管理等进行分析和改进
质量保证	对如何进行产品和过程质量保证进行分析和改进
量化管理	对如何进行研发过程的信息透明、开发度量进行分析和改进
运营管理	改进域描述
服务运营	对如何高效地收集、分析服务用户数据并促进产品优化进行分析和改进
客服管理	对如何高效地响应、统计和分析客户请求进行分析和改进
团队管理	改进域描述
跨职能团队	对如何创建、建设和管理研发团队进行分析和改进
团队培训	对如何开展团队培训提高团队整体战斗力进行分析和改进
过程资产建设	对如何建设团队的过程资产、管理团队知识进行分析和改进
团队协作	对如何开展和确保研发团队的高效协作进行分析和改进

(2) 改进域建模

每一个改进域统一通过以下简单结构进行组织：

- 改进域主要概念和规程介绍

介绍该改进域的主要背景和核心工作流程。

- 现状表述与问题分析

介绍该过程域中涉及的主要问题。

· 改进的切入点和思路

对问题进行分析，并提供解决问题的思路。

· 改进的模式和实践

过程改进中采用的主要方法以及相关工程实践。

伴随改进域的分析、展开和总结，可能会有文档及过程资产等需要梳理，在每个改进域中也会有相应的描述并提供简要的模板说明。

(3) 角色和职责

轻量级过程改进主要围绕研发团队展开工作，同时包括研发团队外围的部门和团队，本文中提到的每个改进域都是站在研发团队的角度上看问题，研发团队角色是其主要的参与角色，但可能也包括一些配合型的、非研发团队角色。研发团队主要角色和职责总结参考表 8-2。

表8-2 研发团队主要角色和职责

角色	主要职责
过程改进小组	根据团队的现状分析、计划和裁剪过程改进模型，并负责在团队中推广、实施具体的过程改进措施。
项目经理	项目管理过程改进域的主要负责人，团队中的项目经理、项目实施人员等构成了项目管理团队，这里抽象成项目经理这一统一角色
产品经理	产品管理过程改进域的主要负责人，团队中的产品经理、需求分析人员等构成了产品管理团队，这里抽象成产品经理这一统一角色
运营经理	运营经理通常和产品经理紧密协作，可能隶属于产品管理团队，但也可以形成独立的运营管理团队，是运营管理改进域的主要负责人
开发人员	泛指团队中所有的技术人员，包括设计人员、编码人员等
测试人员	泛指团队中进行测试工作的人员，这里把测试人员（QC）与质量保证人员（QA）做区分，现实中可能是一人同时担当两种角色
质量保证人员	定期/不定期地开展“过程与产品质量检查”、跟踪质量问题，给出质量改进措施
配置管理人员	通常可以由其他角色进行兼任，主要负责维护中央配置库，并进行配置项、基线等配置管理

2. 过程改进域示例

本小节通过“项目计划”这一特定过程域示例展示我们对过程域进行建模、分析以及提炼工程实践的过程。

项目计划的目的包含两个主要方面，对内是为项目的研发和管理工作制定合理的行动纲领，以便所有相关人员按照该计划有条不紊地开展工作；对外是为客

户提供项目的统一视图，确保所有干系人能够根据计划进行工作配合、进度同步并最终提高客户对项目实施进度的满意度和认可度。该改进域主要阐述在项目计划过程中涉及的主要规程、可能存在的问题、分析并提出相应的改进措施。

(1) 项目计划的规程

项目计划过程涉及面很广，其最核心原则是“信息不对称”，即在客户和研发团队之间形成信息传递的过滤和筛选，确保两者之间的信息不对称从而为项目经理把握项目进程提供缓冲并降低风险。如何进行信息的不对称管理，方法就是把项目计划拆分成两个维度，本文使用面向客户的项目实施计划和面向研发的项目研发计划来表示这两个维度。项目计划与信息传递的示意图如图 8-27 所示。

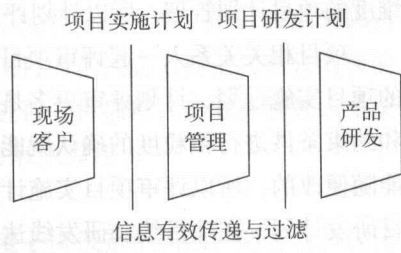


图8-27 信息传递与项目计划

根据上述思路，项目计划通常包括的规程有：

· 制定项目实施计划

项目实施计划面向客户，制定项目实施计划目的在于为项目启动、实施跟踪等提供进度基础，并为销售、市场和运营团队提供统一项目视图。项目实施计划是项目启动会的主要输入。

项目经理主导项目实施计划制定。在制定项目实施计划的步骤上，项目经理主要通过与销售、研发团队负责人的沟通和协调确定项目实施计划，包括项目实施的生命周期划分、里程碑时间、第三方供应商集成等，项目实施计划的依据是项目估计。项目估计的主要工具是粗粒度 WBS 和类比估算，即根据本项目的功能范围通过客户类别分析和以往项目实施经验来得出估算。项目经理根据项目估计制作《项目实施计划》并提交项目管理团队，通过评审之后发布给客户和相关干系人。

· 制定项目研发计划

项目研发计划面向研发团队，制定项目研发计划的目的在于为项目的开发、测试、部署、发布等提供时间依据，并为整个研发团队提供统一项目视图。研发团队负责项目研发计划的制定，项目经理根据项目实施计划推动项目研发计划的形成和完善，注意研发人员眼中的实施计划和项目经理对外的实施计划的不对称性。

项目研发团队根据项目实施计划中的研发阶段进行细化分解，按照研发团队中的研发过程模型（如瀑布、敏捷等）进行时间和人力资源细分。通常按照功能

模块和功能点进行研发任务组织，结合系统开发、集成、测试等步骤进行进度安排并形成《项目研发计划》。项目研发计划的一个重点是系统集成，需要对第三方供应商的研发任务安排有细粒度的计划确保项目实施的顺利执行。

· 评审项目计划

项目计划的评审是为了获取项目对内对外干系人的承诺。由于我们使用两个维度的项目计划管理，所以计划评审也同样采用两种评审模式。

项目相关关系人一起评审项目计划，项目经理负责项目计划达成一致。对外的项目实施计划，计划评审更多是一种计划确认，虽然对项目实施计划中的前置和约束条件进行粗粒度的确认就能启动项目，但面向客户的计划给出去之后是不能随便改的，所以评审项目实施计划应该是计划评审工作的重点；对于对内的项目研发计划，在项目线和研发线达成一致是评审的主要工作，通常对内的计划管理可以比对外的实施计划更加灵活的把握。

(2) 项目计划中的问题

项目计划与研发团队关系密切，一份合理的项目计划无论对内对外都能起到指导作用。但项目计划也是项目管理中最难以把握的环节之一，项目计划过程中可能存在的问题包括：

· 项目计划缺乏两维度分离理念

这是项目计划管理中最大的一个问题，表现为信息过于透明，信息在项目线和研发线之间缺乏过滤机制。作为项目计划制定者所采用的一般策略是：对外我给出去的开发时间是2个月，对内我让研发团队在1.5个月之内完成计划，那我有半个月的时间缓冲，也就是为自己留了一条退路。项目计划管理就是每走一步都能为自己留下一定的退路，这些退路最终会变成一条活路。如果不区分项目实施和项目研发两个维度，信息直接从项目线传递到研发线，就等于在项目经理和研发人员的眼中，开发时间就是2个月，那如果一不小心出现一些突发情况，项目经理就变得没有任何退路。

· 项目实施计划和研发计划混淆

项目实施计划和研发计划混淆也是一个常见问题，表现为实施计划包含过多研发细节，或研发计划包含实施性内容。实际上这两份计划面向的受众完全不同，是两个独立的计划过程，需要使用不同的过程模型和计划方式。项目实施计划基于项目全生命周期进行梳理，而项目研发计划通常只包含项目实施计划中面向研发团队的部分，使用研发过程模型进行梳理。

· 项目实施计划制定缺乏客观评估和评审

项目实施计划面向客户，面向客户的东西都是重要的，都需要过程把控。但现实中往往是面向客户的东西项目经理或管理层不通过团队协作就很武断的做出

决定，这是不对的。项目实施计划的执行需要有相对客观的评估依据和评审流程，因为通过评估和评审，大家就会对计划中的风险和技术陷阱进行提前预判，确保项目实施计划的完整性和合理性。项目实施计划是研发的依据和基础，我们有合理的项目实施计划通常就能得到合理的研发计划。

- 项目实施计划未基于项目实施全生命周期

可能每个项目管理团队对项目实施计划会形成各自的风格，但过程模型都是基于项目实施的全生命周期。全生命周期包括项目的正式启动到正式验收结项的所有环节，这些环节需要根据项目本身的特点进行分析抽象，如服务部署、系统试运行等在互联网应用和企业级应用中可能存在较大差距。

- 项目计划粒度把握不合理

项目实施计划和项目研发计划中的时间粒度上应该是不同的，两者应该保持时间上一个数量级别的差距，项目计划粒度会对后续的项目监控有较大影响。

- 项目计划中缺乏对第三方供应商的管控

项目计划只考虑项目和研发团队自身，而忽略项目实施第三方供应商的参与无疑是致命的。对于系统集成性项目而言，内部研发团队的进度把控往往不会是大问题，但供应商的时间我们很难控制，所以在项目实施计划中明确把他们的时间也加入进来，确保客户、供应商和我们能对系统实施过程中的系统集成环节达成高度一致是项目计划管理的一个难点。项目计划中没有考虑供应商的时间安排，导致研发团队的开发节奏被供应商的不合理计划所打断的情况并不少见。

（3）项目计划的过程改进

项目计划可以说是一个不断变化的过程，因为项目的不同特点很难做到统一的模式，也很难做到计划就等于现实。但我们要有计划管理的方法论，从问题的分析和梳理入手，对项目计划过程改进的切入点包括：

- 关注信息传递过程

在项目计划乃至整个项目管理过程中，我们和客户之间的信息传递都要形成这样一种效果：我们和客户之间有一张纸，这张纸具有一定的透明度，客户能透过这张纸看到纸后面有一个杯子，杯子里有水，但客户却看不清楚这个杯子里到底有多少水。研发团队就是这杯水，而我们的项目计划就是这张纸。项目计划中进行的信息传递就决定了这张纸的透明程度。

- 关注计划分解和粒度

计划的制定基于范围分解，分解的关键在于粒度，上面提到项目实施计划和项目研发计划应该保持时间上一个数量级别的差距，一般前者以周为单位，后者以天为单位是一项比较好的实践。同时项目实施计划由于面向客户，应该围绕项目全生命周期展开分解并确保为后续的项目监控提供合适的计划视图。

向技术管理者转型：软件开发人员跨越行业、技术、管理的转型思维与实践

- 关注团队参与

项目计划的制定需要团队参与，与计划相关的项目、销售、研发、市场、运营等各个角色都应该对计划有明确的认识，其中两种计划的决策者、参与者、知情者都有其获取或制定项目计划的最佳时机。

针对上述切入点，我们梳理项目计划过程改进的模式和实践包括：

- 建立项目线和研发线分离机制

项目线和研发线分离是控制信息传递过程的第一步，项目线和研发线分离之后直接的效果就是两条线有各自的计划，这两份计划之间确保有一定的信息不对称。工作机制上，项目线对于任何一项计划上的决策都应该先于研发线，项目线的决策通过一定的信息不对称处理之后再抛给研发线。在项目管理上，项目线工作的最大难度和挑战莫过于屏蔽来自客户和外部干系人的输入，确保研发线能够集中精力进行系统研发工作。

- 建立项目全生命周期模型和范围分解模板

项目范围分解模板用于指导项目计划中的 WBS 创建工作，对于同类型的项目实施而言，模板都是应该事先去梳理的，无论是项目线上的模板还是研发线上的模板。项目范围模板通常基于特定的项目全生命周期模型进行内容组织，项目全生命周期模板包含了项目实施的各个环节和阶段，是项目实施计划的基础。

- 召开项目计划评审会议

项目实施计划和项目研发计划都需要评审，尤其是项目实施计划，因为项目实施计划一旦定稿往往很难变动，项目实施计划的变动就是项目计划变更。所以项目实施计划的评审通常需要项目线、研发线的负责人或资深人员参与，对于初创型项目计划，建议至少进行初稿、修订稿两轮的评审；如果项目实施过程已经成熟，则一轮评审没有大问题即可通过。如果项目实施计划评审落实到位，项目研发计划评审主要是对实施计划中开发测试部分的分解，一般不会有太大的问题。

(4) 项目计划的过程资产

- 项目实施计划

项目实施计划主要包括以下要点：项目实施全生命周期阶段，从项目启动到项目验收的所有阶段；以周为单位的时间安排，一般以甘特图的形式进行展现；主要里程碑，系统上线等的重要里程碑节点；第三方供应商计划，需要在项目实施计划中以一定形式展现给相关干系人。

- 项目研发计划

项目研发计划主要包括以下要点：系统功能模板，对系统所需开发模块的梳理；开发模型及其表现，对于开发过程模型（如瀑布、敏捷等）在计划中的体现，表现为开发顺序、测试时机、系统集成方案等各个方面；研发人员安排，结合项

目人力资源计划安排研发人员。

- 项目人力资源计划

项目人力资源计划主要包括以下要点：项目管理团队的成员安排；项目研发团队的人员安排。

- 项目范围分解模板

项目范围分解模板主要包括以下要点：标准的项目实施生命周期；标准的系统功能模块，以及对系统功能分解的方法和分解后的结果。

项目计划一方面对项目启动做项目实施计划上的支持，另一方面为后续项目监控和需求管理提供进度管理上的依据。实际上，进度管理是项目管理中最本质也是最可以发挥的部分，项目计划贯穿整个项目管理过程，好的项目计划和差的项目计划对项目实施会造成完全不同的效果，这也是我们要把项目计划作为一个改进域来关注的原因。

过程改进的目标是改进研发团队的整体绩效，过程改进是一项重要和长远的工作，需要根据机构的发展战略、研发实力等实际情况来梳理过程域和改进方案，并要充分考虑过程改进的成本和效益。轻量级过程改进的宗旨是针对没有专设过程改进部门的中小型研发团队，通过比较低的代价有效地改进过程能力，目标是能达到适合团队发展的过程能力。技术管理者作为过程改进的主要推行人员应当具备一定的软件工程和项目管理知识，再掌握主流的软件开发管理模型和过程改进模型，并进行裁剪和扩充。

8.5 本章小结

研发过程体系建设是作为技术管理者开展技术管理工作最重要的环节。对于研发过程体系建设的探讨，本章的思路是先梳理典型的软件过程模型，然后对目前软件开发领域最常见的敏捷方法进行详细展开，再采用过程改进思想来指导日常工作中的管理过程。

对于研发过程体系的理解都可以从“管道理论”出发，因为软件开发管理的对象和目标就是追求资源、时间、成本等通过管道时所应达到的一种平衡。本章对 Scrum、精益思想、看板方法、极限编程等敏捷方法的讨论都是基于管道理论，并结合各个敏捷方法的特征给出最佳的过程管理以及过程改进的方法与实践。

对于任何一个团队都不能对现存的方法和过程进行照搬照抄，而是应该采用裁剪的思路去建立适合自身团队的过程体系。本章最后也对如何进行过程裁剪、如何进行过程资产建设做了分析，并提出轻量级的过程模型与示例。

9 组织管理

组织管理(Organizational Management)是指通过建立组织架构、规定岗位职务、明确责权关系等手段以有效实现组织目标的过程。组织管理是现代企业管理的一个重要组成部分，也是实现组织战略的一项保障。一个公司中常见的人力资源管理、薪酬管理、绩效管理等都属于组织管理范畴，涉及行政、人事、技术等各个团队的日常工作。本章不会对组织管理的方方面面都做详细阐述，而是侧重于技术管理的角度出发，梳理作为一个技术管理者所应具备的对组织管理的认识以及所需的各项技能。

站在技术管理者角度，我们的核心工作是如何高效地管理技术开发人员，以及如何满足组织的战略目标。组织管理的首要切入点是明确技术管理者在组织中所处的定位和职责，关于这一点我们在下一章中会有详细探讨，这里只要明确技术管理者一般会是一个中间层角色，既要面对上面的战略层，也需要面对下面的执行层，当然对于同一层级的其他角色成员也要进行协作。同时，技术管理者还需要进行自我管理。因此，面对大部分组织场景，我们可以将技术管理者眼中的组织管理划分成以下四个重要方面，即向下管理、向上管理、向外管理和自我管理。

图9-1展示了这四种组织管理的各自侧重点。对不同的管理对象，所采用的管理方式自然也应该有所不同。对于下属技术开发人员，理解、领导、激励、考核构成了基本的管理模式。对于上级管理层，重点是了解他们的期望并从结果角度出发管理这些期望。对于外部团队，通常会采取协商和沟通的方式推动工作。而对于技术管理者自身，重在自我提升。

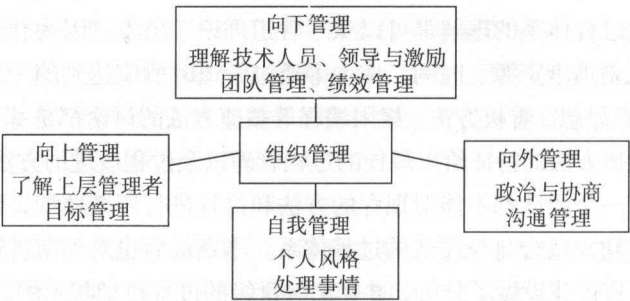


图9-1 组织管理的四个主要方面和各自侧重点

9.1 向下管理

向下管理是组织管理最重要的一个方面，作为技术团队的负责人，通常技术管理者会花费大部分时间在向下管理上。这里“向下”的含义不仅仅是指你的直接下属，在大中型团队中，还可能包含你的间接下属，也就是说技术管理者不仅需要管理下面的人，也需要指导下面的人去做向下管理，从而提高整个团队的工作效率。

9.1.1 理解技术人员

向下管理涉及的内容很多，正确理解技术人员是所有所需开展工作的前置条件。在国内整体销售或运营驱动的开发模式下，技术人员往往处于流程的末端，需要得到更多的理解和认识。很多技术管理工作开展不善就是因为缺乏对技术人员的类型和特点分析所致。

1. 人的四象限模型

本书在很多地方都会使用四象限模型，四象限模型首先需要确定两个维度。对一个人而言，最基本的两大属性维度是态度与能力（如图 9-2 所示）。我们都希望我们的开发人员都能位于第一象限中，既有能力也有好的态度，这种人往往可以将其提升为团队中的管理岗位，使其得到重用。同样，第四象限中的既没能力、态度又不好的人员应该予以淘汰。团队中多数开发人员应该处于态度和能力这两者之间的平衡点，即位于第二象限和第三象限之间，对于有态度但缺少能力的人可以通过培养的方式提升其能力，而对于有能力但态度上有缺陷的人而言，则应采取引导的方式。

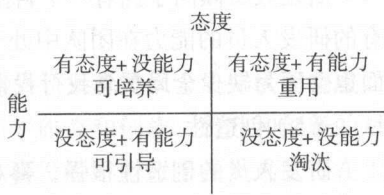


图9-2 人的四象限模型

不同组织、同一组织所处的不同阶段对人的要求显然也会有所不同，软件行业是一个以人为生产核心的行业，针对开发人员技术能力的提升，有效的培养手段通常能够解决能力不足的问题。所以从人员的配备上，我们应倾向于获取第二象限中的初中级开发人员。另一方面，互联网行业，技术更新速度过快，开发人员流转率很高，导致很多组织尤其是初创型互联网公司倾向于直接引入第三象限中有能力的人才并采用引导的方式使其适应组织环境。

2. 人员类型和特点分析

在 1.1.2 节中的分析中，我们已经从技术开发人员与管理人员之间的对比中了解了开发人员的部分特征，可以看到开发人员与管理人员之间存在较大差异，这

也是本书写作的目的所在，即提供从开发人员到管理人员的转型方法。另一方面，开发人员的类型和特点客观存在，作为技术管理者，除了对比与自身工作内容和方式上的差异，我们也需要深刻理解作为开发人员的其他特征，包括：

（1）团队要求

现代软件开发都是团队行为，技术人员对团队成员要求普遍偏高。技术人员所具备的专业性要求团队中的其他如产品、项目等非技术类人员具备与其同样级别的专业性。如果团队中不同工种的专业水平参差不齐，技术人员往往会表现出一定的不协作性。

（2）高敏感性

研究开发工作要求研发人员具有较高的洞察力，能够发现别人所不能发现的问题。这种高度的敏感性也使得技术开发人员在日常的待人接物方面比一般人更加小心谨慎，但是，这种性格有时会影响研发人员在研究开发过程中的相互合作和团队精神的培养。

（3）追逐前沿

知识是研发人员赖以生存的技能，随着行业技术的日新月异，研发人员必须不断学习才能保证自己的技能观念、行为习惯适应技术革新的要求。

（4）自主独立性

研发人员倾向于拥有一个自主的工作环境，重视工作中的自我管理和独立性。有的研发人员的能力在团队中处于领先的状态，一方面希望展现其才华，另一方面也会因为缺少全局观念我行我素、不服从统一管理而影响企业组织关系的协调。

（5）创造性

研发人员的创造性很强，喜欢做以前没有做过的、具有技术挑战性的研究，而不是简单重复。创造是他们体现自我价值和方式和生活追求。

（6）蔑视权威

技能的特殊化和重要性，往往使研发人员对其上司、同事和下属产生影响，对于技术的专注也会使研发人员形成自己意识形态，从而决定了研发人员在企业中的权威影响力。

（7）成就意识强

与一般员工相比，研发人员更在意实现自身价值，并强烈期望得到社会 and 同行的认可和尊重，并不一定满足于完成日常事务，而是力求完美。

（8）高度自尊

研发人员由于受到过良好的教育，从事技术开发工作所要求的独立思考、所面临的来自内部与外部的压力与激烈竞争，使他们同一般人相比更加独立、自尊、自爱。

（9）流动意愿强

新时代背景下，软件开发和产品运营模式对传统的雇佣关系提出了新的挑战，尤其是互联网行业，由于行业本身存在的不确定性远高于传统行业，研发人员因为主观或客观原因导致的流动性非常普遍，研发人员的忠诚感更多是针对自己的专业而不是雇主。

针对以上 9 大特征，我们也可以对开发人员进行分类。分类的依据同样有很多维度，这里列举几种目前在互联网行业中比较典型的维度来分析开发人员类型：

（1）开发 VS 测试

在传统瀑布开发流程中，开发和测试界限明确。但在互联网行业快速迭代的开发需求下，敏捷思想深入人心，开发和测试之间的界限越来越不明显。很多团队开发和测试人员的比例维持在 $n:1$ 的状态，这个 n 可能是 5，也可能 10，甚至于有些团队并没有配置专业的测试人员。这就要求开发人员非常重视测试，在自测的同时还要兼顾整个系统的集成测试、性能测试。同时具备开发和测试能力的复合型人员日渐形成了一种新的类型，与传统的纯开发型人员形成对比。

（2）技术 VS 业务

技术型开发人员和业务型开发人员的界限通常比较明确，尤其在大型团队中，分工决定了开发人员的类型。很多团队会设置专门的中间件、数据库等专业技术团队，用于为业务导向的产品研发团队提供服务。

（3）开发 VS 研发

有时候我们可把“开发”和“研发”两个词分开来看技术人员，从而构成两个不同的类型。开发偏重于实现，而研发则关注于研究和创新。研究和创新的对象不仅仅针对技术，也可以是业务。总体而言，研发的难度大于开发难度，在工作流程中，研发也应该位于开发的上游。

（4）数据 VS 功能

随着大数据相关技术的兴起，面向数据开发成为技术开发的一个重要组成部分。互联网行业普遍偏向运营，而运营的前提是数据支持。区别于普通的功能型开发人员，数据型开发人员则更关注于海量数据之下的处理、分析和挖掘，并通过可视化体系将数据展现给运营等相关决策者。

（5）执行 VS 规划

技术开发通常位于一个组织中的执行层，但技术同样需要规划。从是否对技术发展起到推动作用这一个角度出发，开发人员也可以分为执行型开发人员和规划型开发人员。

从以上分析可以看到，理解技术人员并不是一件简单的事情，即使你自己曾经是一位开发人员也不例外。技术管理者进行向下管理的第一步是理解，在理解

的基础上就可以进行领导和激励，从而使团队中的每一位技术人员都能朝有能力又有态度的人才发展。

9.1.2 领导与激励

管理应该有好的思路和方法论，但期望这些思路和方法论能按照自己想的那样发挥效果，通常只是一种理想。在职权的范围内充分利用人力和客观条件，并以最小的成本办成所需的事情还需要团队负责人的领导力（Leadership）。同时，领导力和激励（Motivation）是两个相辅相成的一因素，对于软件开发这一特定领域而言，领导力最主要的表现或者说最能发挥其作用的切入点是激励。

1. 领导力

业界关于领导力的定义有很多，我们这里引用大师 Gerald M. Weinberg^[38] 的说法。所谓领导力，就是创造这样一个环境，每个人都能在其中发挥出更多的能力。从这个定义上讲，领导力就是催生其他人身上的创造力和生产力。

领导力针对的对象一般也可以分为人和过程，其目的是创造一种特定的环境，对环境中的的人做出反应，向他们提供选择，并给他们一定的自由度。我们对领导力进行梳理，会发现其首先体现在用人上，找合适的人并进行管理是领导的第一步。其次，我们会形成团队的价值观，并希望团队中的所有成员共同遵守，团队负责人在形成这种价值观时会起到主导作用。同时，提出优化流程的建议并付诸实施能显著提升领导力，团队成员会在优化的流程中发挥越大的作用，领导力也就会产生越大的影响力。最后，要注意个人在演讲、聚会、会议上的表现，努力提升个人魅力。

对于提升领导力过程中应该遵循的原则，我们认为信息透明和授权是领导力相关的两条核心原则。

信息透明在这里可以理解为包含自我透明化、项目透明化和关系透明化三层含义。自我透明化指的是表现自身的优点和缺点、承认自身的实力和兴趣并积极参与上下级沟通，让别人了解你是让别人信任你的基础。很多技术人员由于工作环境以及自身性格原因，在自我透明化上存在明显缺陷，难以成为一个技术负责人，或者即使成为团队的负责人之后，因为缺少与团队成员的相互了解导致领导力大打折扣。项目透明化主要体现在让领导看到你的困难，对时间和进度上的透明度做把控，不能太透明也不能不透明。同时，对项目中技术体系进行合理的透明化有助于各种外部干系人对技术团队的了解，提高协商过程中的筹码。创建自身的风格并保持不变，在做事情之前进行有效倾听，同时提供让别人透明化的场景和机会是关系透明化相关的实践方法。

授权往往是技术人员所不擅长的一个领域，很多技术负责人在紧急情况下都

倾向于自己出手，而忘了整个团队。身体力行，让别人看到你在做事情是提升领导力的一种方法，但在有些场景下，通过授权让团队成员去完成有难度的任务恰恰更能提升领导力。建立信任关系是授权的第一步，需要在平时进行不断经营。而对某个具体场景，在授权之前确保团队成员与团队负责人达成共识。

信息透明化的具体实现可以借助于一些工具来展现可视化信息，而授权的切入点在于使问题简单化。无论采用各种原则，尝试在团队中推销自身的想法，并对核心问题和痛点保持关注。

领导力的表现形式有很多种，如带队育人的教导力、合理分配资源的组织力、高瞻远瞩的决策力、人心所向的感召力等，技术管理者自身能够快速成长的能力也是领导力的一种表现。

2. 激励

(1) 激励理论

业界关于如何进行激励存在一些主流的认知体系，尽管这些体系偏于理论，但作为技术管理者而言，仍有必要充分理解并进行针对软件开发领域的思考 and 理解。激励的主流理论包括：

· 马斯洛需求层次理论

马斯洛需求模型^[39]分为五层（见图 9-3），处于底部的生理需求和安全需求无须展开，社会需求是指融洽的工作和交往环境，尊重需求体现在对工作的认同以及重视沟通和协商，而自我实现更多表现在渴望被分配有挑战的工作以及培训。对于普通开发人员而言，更容易受发展机遇、个人生活、成为技术主管的机会以及同事间人际关系的影响，不容易受地位、受尊敬、责任感、与下属关系以及认可程度的影响。影响工作动力的工作特征因素包括技能多样性、任务独特性、任务的意义以及是否提供了积极正面的反馈。

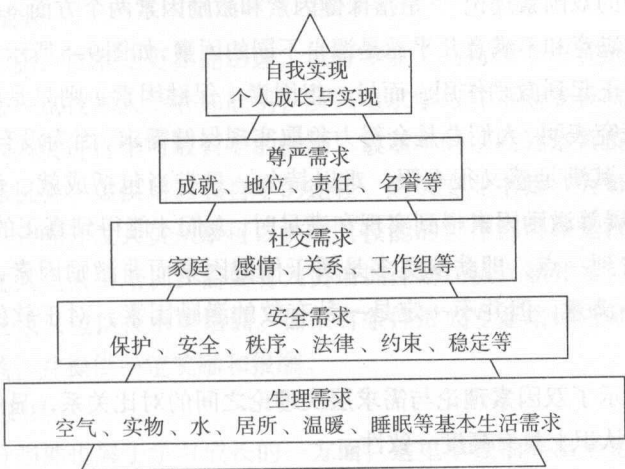


图9-3 马斯洛需求模型

· 麦格雷戈的 X-Y 理论

麦格雷戈的 X-Y 理论^[40]是一对基于两种完全相反假设的理论，X 理论认为：多数人天生懒惰，尽一切可能逃避工作；多数人没有抱负，宁愿被领导批评、怕负责任，视个人安全高于一切；对多数人必须采取强迫命令，软硬兼施的管理措施。Y 理论的看法则相反，它认为，一般人并不天生厌恶工作，多数人愿意对工作负责，并有相当程度的想象力和创造才能；控制和惩罚不是使人实现企业目标的唯一办法，还可以通过满足职工尊重的需要和自我实现的需要，使个人和组织目标融合一致，达到提高生产率的目的（见图 9-4）。X-Y 理论的实施与一个团队乃至一种文化的特点有直接关系。在国内，X 理论根深蒂固，在技术团队中充分发挥 Y 理论的优势可能是实现激励的一种有效手段。

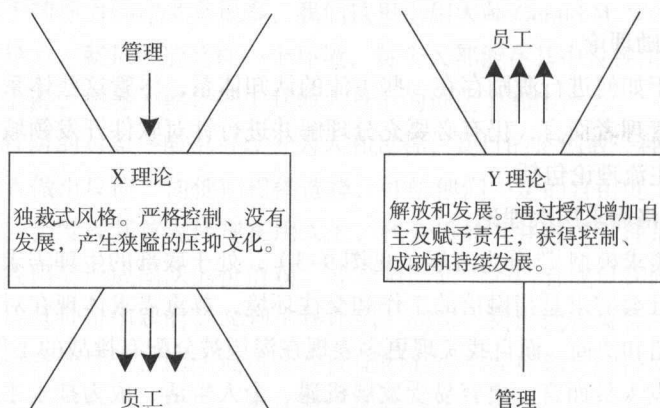


图9-4 麦格雷戈的X-Y理论

· 赫茨伯格的双因素理论

赫茨伯格的双因素理论^[41]是指保健因素和激励因素两个方面。双因素理论展示了工作中的满意和不满意几乎总是源自不同的因素,如图 9-5 所示,一组因素(激励因素)能真正起到激励作用,而另一组因素(保健因素)则只是消除不满意。赫茨伯格的研究表明,人们总是会努力争取实现保健需求,因为没有达到会不满,而一旦得到,其满足感又很有限,难以持久。只有当包括成就、认可、工作本身、责任和发展等激励因素得到实现和满足时,人们才能得到真正的激励。从 9-5 中我们可以看到一点,即薪酬更多是属于保健因素而非激励因素,所以金钱只能用于消除不满意,但并不一定是一种有效的激励因素,对于软件技术人员尤其如此。

图 9-6 展示了双因素理论与需求层次理论之间的对比关系,显然两者之间对于激励因素的认识上具有高度一致性。

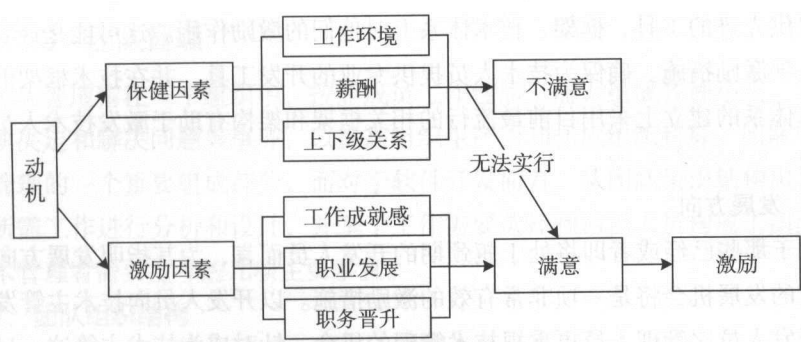


图9-5 赫茨伯格的双因素理论

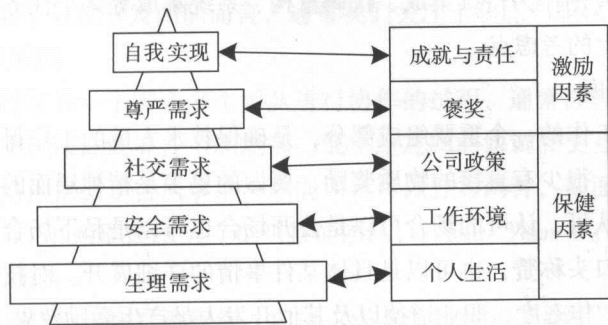


图9-6 双因素理论与需求层次理论的对比关系

(2) 激励措施

掌握激励理论的目的是为了实施激励措施。因为基本的激励理论面向所有人，所以我们需要针对软件开发人员做进一步分析才能获取相应的激励措施。针对软件开发人员，我们认为典型的激励因素包括以下几个方面：

· 学习成长

在软件行业，技术变更如此迅速，没有专业上的持续学习和成长，就可能很快被淘汰。这一点技术人员心知肚明，所以追求学习并不断成长就成为首要激励因素。对技术人员进行学习成长上的激励，最基本的手段就是营造良好的学习环境。提供进修机会、提供培训和自学的途径、购买专业书籍、为每个新手开发人员指定导师、分配开发人员从事可以扩展其技能的工作都可以是行之有效的工作设计方式。尤其是人员培训，需要有人员培训计划。从入职开发，提供全生命周期培训，不应该只包括技术类培训。如果有条件形成专业培训讲师团队，对培训讲师进行考核，并提供一定奖励和报酬。

· 技术工具

技术提升当然也属于学习成长的一方面，这里单独抽离出来是想强调为技术

人员提供先进的工具、框架、技术体系上对他们的激励作用。这可能是最容易做到的一项激励措施。确保为技术人员提供专业的开发工具，并在技术框架的选型和技术体系的建立上采用目前最流行的相关框架和架构有助于激发技术人员的开发热情。

· 发展方向

对于那些已经或者即将处于瓶颈期的开发人员而言，为其指明发展方向并提供相应的发展机会将是一项非常有效的激励措施。以开发人员向技术主管发展为例，开发人员比管理人员更重视技术管理的机会，针对成为技术主管这一目标，指派每个人分别作为业务模块、数据库、报表等某个特定领域的技术负责人，或者指派每个人分别作为持续集成、代码重构、系统集成等某个任务的技术负责人都是工作设计上的考虑点。

· 认可程度

技术管理工作的一个重要组成部分，是确保技术人员的工作得到认可。技术开发不像营销，很少有直接的物质奖励，要做的更多是精神层面的激励。技术人员内心渴望被认可，认可的场合可以是公开场合也可以是私下场合，认可的方式可以是简单的口头称赞，也可以是具体某件事情的详细展开。对技术人员工作的认可能够对其工作态度、职业道德以及其他开发人员产生激励效果。

· 工作环境

工作环境对技术人员的激励作用在现在的互联网行业表现非常明显。无论是灵活打卡或不打卡制度、加班补助制度，还是上文所提到的学习环境等，都对技术人员有良好的激励作用。这些激励因素往往是一种相对的附加效果，没有属于正常，但有的话就能让技术人员高度认可并更加容易融入工作环境中。

· 直接利益

直接利益一直是个人绩效的重要因素，传统意义上的利益就是指工资、奖金、股票、期权等。高工资和可能拿到的高奖金是最基本的激励因素，但这需要建立在个人以及团队绩效的基础之上。

· 管理风格

团队管理者的管理风格可以说对开发人员的工作表现有至关重要的影响。管理风格的一大表现形式就是对技术的重视程度，技术人员通常希望得到技术上的尊重并希望有一定的自主权，当开发人员为实现自己设定的目标工作时，会比为别人更加努力的工作，这就是从成就感的角度来设定激励目标。体现在工作设计上，项目中的开发进度计划应尽可能让开发人员自己把握是一项可以实施的激励措施。

9.1.3 团队管理

团队管理指在一个组织中,按照成员工作性质、能力组成各种小组,参与组织各项决定和解决问题等事务,以提高组织生产力和达成组织目标。团队管理是组织管理的一个重要组成部分,而对于软件开发而言,从团队组织结构出发,对团队所需工作进行分析和设计,并基于工作需要选择和培训人员构成了团队管理中技术管理者需要主导的几项主要事宜。

1. 团队组织结构

实际上,全世界企业的团队组织结构就这么几种,每一种都有优点和缺点,没有一种组织结构是完美的,而且很多团队组织结构是一种混合体,即由多种不同维度组成。对于软件开发团队而言,通常我们关注于职能和职权两个问题。

(1) 按照职能

软件项目研发是一个需要多个团队进行协作的过程,通常涉及项目线、产品线、技术线、质量保证线等各个职能部门或小组之间的协调和交互。软件开发团队的团队组织结构根据是否从事单一职能而言可以分为两种,即面向职能性团队(Function Team)和跨职能团队(也叫特征团队, Feature Team)。

· 职能型团队

职能型组织结构亦称 U 型组织,即以工作方法和技能作为部门划分的依据。在软件研发过程中,产品、项目、开发、测试、运维等各个角色都可以形成独立的职能团队。因为业务活动都需要有专门的知识 and 能力,通过将专业技能紧密联系的业务活动归类组合到一个团队内部,可以更有效地开发和使用技能,提高工作的效率。

职能型团队的最大问题在于没有一个直接对项目或产品负责的强有力的权力中心或人,各个职能团队之间因为没有形成统一的目标导向,其协调过程十分困难。

· 跨职能团队

而对于跨职能团队而言,在团队内部应该包括产品、开发、测试、运维等各种角色,能够完成某个产品或服务的整个生命周期,组合职能型团队与跨职能团队的组织结构见图 9-7。

跨职能团队是一种有效的团队管理方式,它能使团队内(甚至团队之间)不同领域员工之间交换信息,激发产生新的观点,解决面临的问题,协调复杂的项目。但是跨职能型团队在形成的早期阶段需要耗费大量的时间,因为团队成员需要学会处理复杂多样的工作任务。在成员之间,尤其是那些背景、经历和观点不同的成员之间,建立起信任并能真正的合作也需要一定的时间。

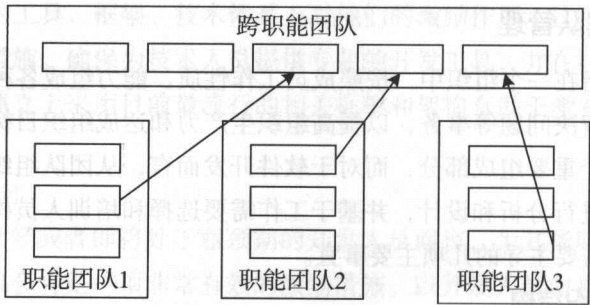


图9-7 组合职能型团队与跨职能团队组织结构

(2) 按照职权

我们知道无论是传统型企业还是新型互联网应用，软件开发过程都需要遵循第7章中所介绍的项目管理流程。从项目管理角度而言，团队组织结构一般都会使用矩阵型结构，而矩阵型结构根据项目经理的职权大小又可以把矩阵型团队归为强矩阵、弱矩阵和平衡矩阵团队。这三种矩阵团队组织的特点参考表9-1^[37]。

表9-1 三种矩阵式组织结构

	弱矩阵	平衡矩阵	强矩阵
项目经理的职权	小	小到中	中到大
项目经理的角色	兼职	全职	全职
项目预算控制权	职能经理	混合	项目经理
可用的资源	少	小到中	中到大

弱矩阵组织结构基本保留职能组织结构的大部分主要特征，但在组织系统中为了更好地实施项目，需要建立相应明确的项目管理团队。

平衡矩阵组织结构是对弱矩阵组织结构的改进，为强化对项目的管理，在项目管理团队内，从职能部门参与本项目活动的成员中任命一名项目经理。项目经理被赋予一定的权力，对项目整体与项目目标负责。

强矩阵组织结构具有项目的线性组织结构的主要特征。强矩阵组织结构在系统原有的职能组织结构的基础上，由系统的最高领导任命对项目全权负责的项目经理，项目经理直接向最高领导负责。

针对软件开发，跨职能团队可以用于消除在信息传递过程中为了确保信息有效性而产生的浪费。跨职能研发团队成员主要包括项目线、产品线、技术线、质量保证线等，大家围绕某一个产品线/平台开展所有工作，确保坐在一起并能够实时、面对面沟通。互联网开发环境下，跨职能研发团队还可能包括运营、客服等角色，但销售、市场等相关人员通常不在该团队中。这实际上是一种强矩阵的

团队组织结构，所有人保持在同一认识水平和工作节奏中。

在内部团队中，根据团队目标组建专门进行日常开发、解决问题、技术创新，或成立专门的专项小组对业务或技术进行短期突破都是团队负责人可以考虑的团队组织结构方式。组建方式上，也可以有不同的切入点，如面向技术体系的分解，基于这种组建方式，人员职业发展应面向技术，可以交流新技术和新思想，但不同团队之间可能存在交流问题；也可以采用面向任务的分解，从完整的项目结构出发，把握功能模块；或者采用面向生命周期的分解，需求、开发、测试、运维由不同团队完成；而如果团队基于矩阵结构组建，则团队成员会有面向项目和面向技术两个主管，需要同时考虑汇报关系和平衡性。

2. 工作分析

工作分析（Job Analysis）又称职位分析或岗位分析，是指全面了解、获取与工作相关信息的过程。是对组织中某个特定岗位的工作内容和职务规范的描述和研究过程，即制定工作描述（Job Description）和工作规范（Job Specification）的系统过程。对于软件行业而言，技术管理者进行工作分析的作用十分明确，即通过工作分析选拔和任用合适的开发人员，制定有效的团队发展计划和人员预算，并为人员培训和绩效考核提供依据。

工作分析涉及两个主要方面，即工作本身和人员特征。系统化的工作分析应该包括以下内容：

- What：要做什么事，表示内容
- Why：为什么做，表示目的
- When：什么时候做，表示时间
- Where：什么场所做，表示地点
- How：怎么做，表示方法
- Skill：需要什么能力，表示技能

以上内容的获取一般可以采用如图 9-8 所示的工作分析流程。准备阶段的主要任务是了解基本情况，确定工作分析的调查方案并明确调查方法。调查阶段是对整个工作过程、环境、内容和人员等各个方面做一个全面调查。分析阶段则对各种不同岗位的工作特性和人员特性的调查结果做全面分析和总结。应用阶段的主要任务则是根据收集和分析的结果，完成工作描述和工作规范。

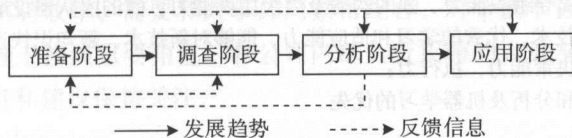


图9-8 工作分析的流程

整个工作分析的流程一方面与上一小节中介绍的团队组织结构有非常直接的关系。技术管理者对工作分析所开展的所有内容都是围绕技术团队组织结构所展开；另一方面，工作分析的目的和结果也与组织的人力资源管理中人员招聘、绩效评估、薪酬设计等密不可分。

工作分析的结果是形成工作说明书，包括工作描述和工作规范两个部分。工作描述就是说明工作本身有关的特征，而工作规范规定的是对从事该工作的候选人的具体要求。工作描述的具体内容因团队而异，但一般都会采用类似的表现形式。如下文即是根据团队发展需要进行工作分析并得出的一份关于公司 CTO 的工作描述，包含了职务描述、职位要求、汇报对象、下属人数等常见信息。CTO 显然是一种技术管理者角色，可以看到本书中的很多内容与该份工作描述一致。

职位描述：

1. 对企业的长期发展负责，制订有关技术的愿景和战略，把握总体技术方向，监督技术研究与发展的活动，并对技术选型和具体技术问题进行指导和把关，完成所赋予的各项技术任务/项目
2. 执行并监督实施技术的长短期战略，组织新技术研发应用，参与并保证对客户需求的满足，参与知识产权策略制定
3. 作为技术方面的权威，从技术角度非常有效地帮助公司推广理念，其中包括公司对技术趋势所持的看法，要对公司下一步的技术发展方向进行一些研究、探讨，做出判断并帮助CEO做出决策
4. 负责技术团队建设和管理，持续提升团队工作、质量和效率。协调部门内外部信息的及时的沟通，有效沟通业务、职能部门，提供技术服务支持。负责技术队伍的建设，做好人员配置与协调，有效监控项目进展；负责所辖员工的培养选拔、业绩考核、工作分配与协调等日常工作；
5. 其他上级交办的任务。

职位要求：

1. 本科及以上学历，计算机、软件相关专业；
2. 10年以上互联网行业开发经验，其中3年以上技术管理相关经验；
3. 有移动医疗系统、互联网行业从业背景或有在BAT工作者优先。
4. 熟知现代企业管理，了解战略制定、运作管理、人事及财务管理知识；
5. 较强的战略性思维、创新意识、执行力及分析判断能力，卓越的表达沟通能力，数据敏感、人际敏感，擅于发现并解决问题；
6. 熟悉商业流程分析与设计；对技术具有高度的敏锐度，能够从客观和业务的角度来进行技术实现，并能及时掌握市场发展动态，对公司技术发展提供决策性的建议；具备大型系统平台的软件开发、平台建设、项目管理经验，有公司整体技术发展方向前瞻性的规划制定和落实及评估的工作经验
7. 具有独立工作及领导组织能力、强的沟通及组织协调能力和强的团队建设能力及影响力，培养和提高团队整体技术，优秀的学习和适应能力，能够对新技术、新知识快速掌握，较好的沟通能力，理解力，决策能力，执行力；
8. 精通大数据挖掘和分析及机器学习的优先

汇报对象：CEO

下属人数:50人

3. 选择人员

在软件大师 Frederick Brooks 的著作《人月神话》中提到,在拥有两、三年工作经验的开发人员当中,在生产力上好坏之间要相差最多 10 倍^[30]。这是二十几年前的一项研究成果,放到现在来看也是非常正确的。正如我们在 9.1.1 中分析的那样,开发人员可能是各种类型的人,所以发现和选择优秀的人员无疑是一项挑战。

(1) 候选人分析

技术管理者的一项日常工作就是招到合适的开发人员。我们通过工作分析已经有了工作描述,选择人员就是要做到人岗匹配。匹配度的衡量来自于对职位分析、团队分析以及企业环境和文化分析。对任职者能力的要求、与其他岗位的关系、主要业绩考核标准等都属于职位分析相关的内容。针对团队分析,团队一致性与互补性、人员个性与团队特点的匹配度是关注重点。而企业环境和文化分析,需要考虑团队以及公司级别的政治因素。

根据 MBTI 职业性格测试,大多数开发人员偏理性和内向,不太愿意进行社会交往。换句话说对于很多技术管理类岗位,技术专家并不一定是好的研发经理,这就需要团队负责人在选择团队中高层人员过程中特别加以注意。另外一个需要注意的点在于找合适而不是合格的人员,所谓合格的人员指的是在简历上显示具有丰富工作经验的候选人员,而合适的人员则是指真正能够干好该项工作的人员,合格的人员不一定就是合适的人员。

(2) 招聘过程

招聘是一场持久战,尤其如果你是一名中小型公司的技术管理者,不要指望招聘信息一发出去,马上就能找到合适的人选。如果是在一个创业公司,则可能会发现,合适的人员很可能来自你或周围同事的人脉。

内部推荐也是目前受欢迎的一种招聘方式,对于那些并没有太多经费去请专业猎头的公司而言,这可能也是最省钱、最有效率的一种方式。通过团队内部人员推荐的途径,可以让团队中现有的人来推荐他们的朋友或以往的同事。显然,这里的内部人员首先需要是靠谱的,靠谱的人所推荐的人一般也是靠谱的,这是我们的经验。雇员推荐唯一需要注意的是必须避免在团队中形成帮派文化,即在推荐的人数上不宜过多。

为了提高招聘效率,需要和人事部门紧密配合。对于技术管理者而言,人事部门在团队管理上扮演重要角色。和公司内部负责招聘的人员友好相处,对他们提供的反馈信息和建议保持关注。

(3) 面试技巧

面试需要技巧。面试是一个双向的过程,其目的是想让对方加入我们,而不

是单纯的评价对方的水平。在公司能提供的条件之外，面试的过程很大程度会影响到对方的判断。对于中小型企业而言，找到合适的人不容易，就更需要技术管理者在面试过程中采用合适的方法。

面试实际上是双方了解彼此的过程。因此首先要摆正自己的位置，不要给人盛气凌人的感觉。同时，尊重对方意味着面试前要做好功课，大致了解一下对方的简历。很大技术人员比较内向或者少言寡语，这就需要面试过程进行破冰，让面试者尽快进入状态。在面试开始的时候，简单的介绍自己、目标公司或团队状况有利于引出对方的兴趣点，从而展开话题。

有不少面试官会在面试中不停地发问。这种方式看似十分主动，但其实不一定能从面试者身上得到有效的信息。比较有效的方法是让面试者自己多讲，面试官一边倾听，一边根据情况提问，引导并控制面试者话题。

无论面试结束你是否已经做出了录用或不录用的决定，都要给面试者一个提问的机会，而且要认真应答。面试是双向的过程，如果你希望面试者能接受这个机会，那么这就是你说服他们的时候。就算你不录取他们，但你希望他们能对你和公司留下好印象，也许可以帮你推荐更多的人，也许他们改进了以后还会回来。千万不要低估他们的口碑对公司造成的影响。

4. 培训人员

从员工培训的类型上讲，一般存在员工技能培训和员工素质培训两种。前者是针对特定岗位，对开发人员进行的岗位能力培训。后者则是对人员素质方面的要求，主要有心理素质、个人工作态度、工作习惯等的素质培训。针对人员素质方面的培训一般会由公司人力资源主导，作为技术管理者，培训人员的主要需求来自于员工技能的提升，在 9.1.1 介绍的技术人员特性以及 9.1.2 中介绍的人员激励都提到需要把培训作为一项日常工作。

除了作为一项激励措施，培训还能够提高企业素质，从而使其能够适应企业外部环境的发展变化，并提高开发人员的工作绩效。关于绩效，我们将在下一节中展开讨论。在这里，我们认为培训的主要目的在于让开发人员统一思想和工作方式（见图 9-9）。这对软件开发而言至关重要，因为软件开发工作由于其特殊性往往很难做到这一点。

对于技术技能类培训，需求比较明确，主要考虑的对象是培训开展的方式。包括培训的分类和培训的方法。

按时间期限划分，培训可以分为长期培训和短期培训，长期培训一般计划性较强，有较强的目的性。按培训方式，又可分为在职培训和脱产培训两种，技术培训一般都会采用在职培训。按培训体系，可划分为组织内培训体系和组织外培训体系两种。组织内培训体系可以包括针对同一个开发技能或工具的集中式培训，

也可以包括针对某一个人所开展的个别培训，当然，固定节奏和频率的日常培训也是很常见的一种开展方式。组织外的培训体系则更加多样化，参加如 QCon 等业界主流的技术研讨会议，或者参加由知名大公司举办大型交流会议等都比较常见。

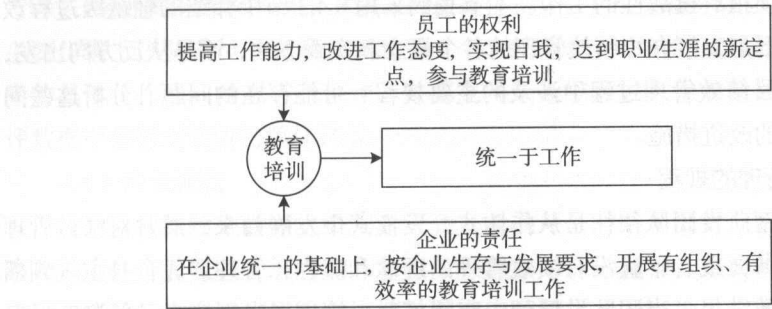


图9-9 培训的目的和作用

按开展培训的方法，最基本的应该是讲授法。其优点是运用起来方便，便于培训者控制整个过程。缺点是单向信息传递，反馈效果差。常被用于一些理念性知识的培训。网络培训目前在在线教育领域应用广泛，优点是运用视觉与听觉的感知方式，直观鲜明，但同样面临着反馈效果差的问题。另外，也存在如研讨型的讨论法、针对某个特定具体问题的案例分析法、针对某个特定场景的角色扮演法、基于导师制度的个别指导法等多种方法可以使用。技术管理者对这些培训方法的应用一般也比较直接，基于需求可以综合以上方法开展培训。

9.1.4 绩效管理

作为向下管理的最后一环，绩效管理是对团队成员进行工作评估和激励的过程。事实上，绩效管理与向下管理的工作分析、员工激励、员工培训等多个方面都有联系，图 9-10 展示了这些工作之间的相互关系。

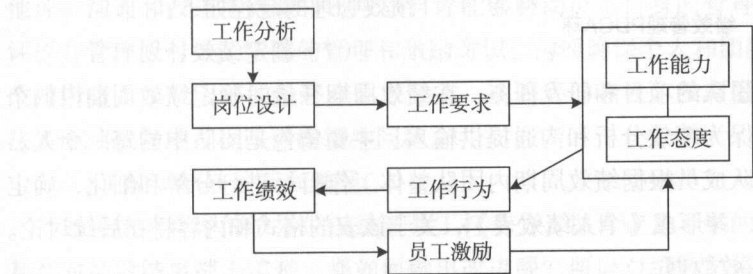


图9-10 团队管理、员工激励与绩效管理之间的关系

虽然很多时候会由人事部门主导员工的绩效管理，但对研发团队而言，技术人员的绩效管理很难把控，所以很多团队往往对绩效管理避而远之，采用管理层主观判断的方法进行绩效把控；有些团队虽然会做一些绩效管理，但只是关注于绩效考核，而忽略绩效背后的工作计划、评估、激励以及过程改进。研发团队的绩效管理是一项很有挑战性的工作。本节我们采用 8.4.3 节中介绍的轻量级过程改进的思路，并不是直接介绍绩效管理的各个概念和实践方法，而是从反方向出发，主要阐述在项目绩效管理过程中涉及的主要规程、可能存在的问题、分析这些问题并提出相应的改进措施。

1. 绩效管理的规程

国内中小型研发团队往往是从作坊式开发模式中发展而来，通常对绩效管理的意识比较淡薄，或者干脆没有绩效管理的理念和流程，管理层凭自身主观判断确定员工的绩效结果。当团队发展到一定规模时，管理层发现靠自己的判断已经不行了，所以就要搞一下绩效管理。这时候的绩效管理可以理解为是团队需要进行过程改进的一种预示。我们都知道过程改进领域有一个 PDCA 环，而绩效管理本身实际上也是一个 PDCA 环（见图 9-11）。

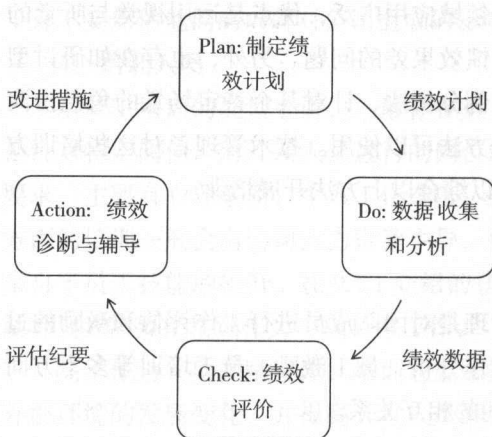


图9-11 绩效管理PDCA环

过程改进如同项目计划需要进行阶段性规划和控制，绩效管理也是一样。通常绩效管理具有周期性，即以一段时间为限形成上面的PDCA环，实际操作过程中以一个月或一个季度作为基准的情况较多，最好不要超过一个季度，否则PDCA环的时效性将大打折扣，下文统称这一周期为绩效周期。结合绩效管理的PDCA环及其周期性，绩效管理的规程如下：

（1）制定绩效计划

目的是根据团队的项目和研发任务，在绩效周期开始时确定绩效周期内的个人工作计划，确保为绩效分析和沟通提供输入。主要角色是团队中的每一个人。主要步骤上，团队成员根据绩效周期内团队整体工作目标进行分解和细化，确定个人的绩效计划，并形成《个人绩效表》，关于该表的格式和内容将在后续讨论。

（2）收集绩效数据

目的在绩效周期结束时根据个人在绩效周期内的工作情况，收集绩效数据并形成绩效结果，为绩效沟通提供个人的自评。主要角色同样是一个人。主要步骤上，

团队成员基于在绩效周期初确定的《个人绩效表》中的绩效计划，结合绩效周期中的具体工作完成情况进行自评，并填充《个人绩效表》中的自评绩效数据。

（3）评估绩效

目的在于根据绩效周期内的绩效数据以及团队整体计划对团队成员的绩效进行评估，从而明确绩效结果，为绩效沟通提供他评。个人直属上级会主导这一过程。主要步骤上团队成员的直属上级基于《个人绩效表》中的绩效计划以及绩效周期中的具体工作完成情况进行他评，并填充《个人绩效表》中的他评绩效数据，他评数据中包括对绩效的整体评分。

（4）沟通绩效

目的是根据绩效计划、员工自评和上级他评，对结果进行分析并明确改进思路 and 措施。个人及其上级会和人事专员一起参与这一过程。主要步骤为个人及其直属上级进行面对面沟通，对该绩效周期内的结果展开讨论，主要针对其中存在的问题触发团队成员自身的思考并找到改进的切入点。

2. 绩效管理中的问题

（1）混淆绩效管理和绩效考核

绩效管理和绩效考核是两回事，绩效考核应该是绩效管理中的一个环节，结合上文中的绩效管理 PDCA 环，绩效考核通常只包括 Plan 和 Check 两个环节，多为事后进行评估，注重形式和结果，主要是人事部门参与整个流程；而绩效管理根据团队目标设定绩效期望值，设定绩效指标后，不断激励并辅导员工，注重整个管理流程，通常团队管理者参与整个过程。片面强调绩效考核而不关注整体绩效管理流程对员工自身的提升非常不利。

（2）没有应用 PDCA 环进行绩效管理

对过程改进而言，绩效管理需要形成完整的改进闭环，我们提倡的就是 PDCA 环。但很多时候绩效管理往往难以形成闭环管理，闭环管理需要制定绩效计划、数据收集和分析、绩效评价和绩效诊断与辅导这四个环节，绩效的自评、他评、沟通和改进措施缺一不可。自评能够触动员工自身的管理和改进意识，他评提升管理层对员工绩效的管理和激励意识，沟通确保个人和团队之间达成一致，改进措施是本次闭环的最终产出以及下一次闭环的输入。没有进行闭环管理是无法实施绩效管理过程改进的根本问题。

（3）缺少绩效诊断和辅导

如果绩效管理的结果仅仅是对员工打个分，那肯定是不够的。有些时候员工甚至对绩效结果都不清楚，那如何做出改进呢？所以对绩效结果我们需要进行诊断，找出好的地方和不好的地方，对好的地方要保持，对不好的地方要改进。改进的方向和思路通常都需要辅导，因为对普通员工，尤其是新员工而言普遍缺少

过程改进的意识和方法，这时候通过沟通进行绩效的诊断和辅导就变得非常重要。

（4）绩效激励措施不完善

绩效激励措施不属于过程改进的内容，但在绩效管理实施过程中必不可少。激励措施可能是物质上的奖励，也可以是精神和思路上的梳理和鼓励，无论哪种手段，充分的沟通是确保绩效激励的最基本方式。缺乏激励会导致过程改进流于形式，导致团队成员对绩效管理的积极性下降和过程改进意识的淡薄。

（5）缺少从团队的角度管理绩效

如果团队成员较多，通常会进行梯队式管理模式，即将大团队分组管理。对于分组后的各小组而言，绩效管理应该以小组为单位进行，这里的小组就相当于一个团队，上面提到的各项绩效管理规程都可以直接应用。反之，如果在规模较大的团队中还是执行点对点的绩效管理，对于团队管理和个人绩效的提升往往都不是很现实。

3. 绩效管理的过程改进

绩效管理的本质是为了提高绩效，提高绩效当然不能只靠引入一套绩效管理流程就能起到立竿见影的效果。但没有绩效管理的理念，不把绩效透明出来作为个人和团队的一项日常工作，绩效提升也就无从谈起。同时，绩效管理也是个人与团队管理者之间的一种纽带，促使个人和团队之间形成一种沟通机制。绩效管理过程改进的切入点包括：

（1）关注团队级别绩效管理

上面提到绩效管理中的一个问题是“缺少从团队的角度管理绩效”，关注团队绩效也就是说我们要站在团队角度看问题。对研发团队而言，一个团队中包含多种角色，而研发目标通常是一个进度要求，这个进度要求需要团队成员协作才能完成。从过程改进的角度看，个人绩效的提升是一个点，而团队绩效的提升才是一个面，点的提升也是为了面的提升。

（2）关注绩效表现形式

绩效管理的过程和结果需要有合适的表现形式，也就是说好的绩效管理应该具备统一的、合理的模型。目前主流的如 KPI（Key Performance Indicator，关键绩效指标）、BSC（Balanced Score Card，平衡计分卡），还有类似 google 的 OKR（Objectives and Key Results，目标和关键成果）等都是绩效模型。

（3）关注绩效的确认和沟通

绩效管理的 PDCA 环中最重要的就是绩效沟通环节。Plan、Do 和 Check 都是为了最后的 Action 做铺垫，过程改进的目标和措施也正是通过绩效的确认和沟通才能得到明确和推动。当然，绩效沟通需要一定的技巧和方法，确保个人和团队都能从过程改进的角度去看绩效管理。

针对上述切入点，我们梳理绩效管理过程改进的模式和实践包括：

（1）团队目标和计划同步

绩效管理的起点是绩效计划，所以我们第一步就是同步绩效周期内的团队目标和计划。团队的计划一部分如同项目管理中的计划一样，需要根据具体的项目进行过滤和透明并形成统一视图。实际操作中，项目日历通常是项目级别计划同步的一项有效实践。另一部分属于项目以外的工作计划需要根据具体团队的情况进行梳理。

（2）团队分解和开展团队绩效管理

对大团队的绩效管理首先需要进行小团队分解，每个小团队有一个 Leader，这些 Leader 负责自己团队中所有成员的绩效管理以及对应的过程改进，同时这些 Leader 的上一层管理者负责这些 Leader 的绩效，以此类推。每个小团队的人数控制在 10 人以内，建议根据跨职能团队组建原则进行团队分解并维护各自的绩效计划和目标。

（3）应用 KPI 体系

建议在绩效管理中应用 KPI 体系，相对其他绩效模型，KPI 容易理解和上手，使用也比较广泛。KPI 系统的核心是建立 KPI 指标库并确保每个 KPI 能够进行量化，通常我们会根据不同的角色设计不同的 KPI 指标，对于技术人员而言，常见的 KPI 指标有计划完成率、产品研发 BUG 率、线上事故数量和影响等；而对测试工程师，包括计划完成率、工作质量考核等。KPI 的特点是量化，但有些工作不一定能做到量化，不能量化的工作可以归为“重点工作事项”。重点工作事项可以根据具体情况进行设定，例如在本节的上下文中，把过程改进措施作为重点工作事项就是一项最佳实践。

（4）个人绩效沟通和改进方案诊断

确保对团队中每一位成员进行个人绩效沟通，这是触发个人过程改进的最有效时机。个人绩效沟通由团队 Leader 主导，基于但不要局限于《个人绩效表》中的内容。对研发团队而言，研发人员普遍不善于沟通和表达自己的想法，团队 Leader 需要有一定沟通技巧让大家把心里话说出来，同时能够结合团队的整体改进目标和方式以及每一位成员的个人想法为其提供个人过程改进的思路和指导。关于个人软件过程（Personal Software Process, PSP）相关思想和实践方式可参考过程改进大师 Watts Humphrey 的相关著作^[19]。个人绩效沟通的时间不限，如果能够和团队成员进行定期 / 不定期深入沟通，对于沟通时间上的投入性价比是非常高的。

（5）绩效统计和分析

绩效管理是为了过程改进，但毕竟还是要有结果，不然无法体现过程改进的

效果，也无法进行激励。这就需要对个人的绩效进行打分，打分的方式也不外乎基于某种分制的一个分数，或者 ABCD 中的一个等级，打分的依据参考 KPI 和重点工作事项的权重进行计算。关于绩效打分不要太量化，从表现形式上等级的效果会比分数好一点，也比较符合过程改进中的等级提升理念。有了每个绩效周期的绩效结果，我们就可以基于这些结果进行一定时间范围（半年或一年）内的绩效统计和分析，应用统计分析的工具和方法可以得到个人的绩效改进趋势图，奖惩措施也可以基于这些数据进行客观的判断和执行。

个人绩效表是绩效管理中的主要过程资产，包括：KPI 指标，应根据不同的岗位和角色设置不同的 KPI 指标；重点工作事项，包括过程改进的切入点或其他无法量化的重要工作；员工自评，个人对 KPI 指标和重点工作事项的完成情况的总结；员工他评，Leader 对团队成员 KPI 指标和重点工作事项的完成情况的总结；绩效结果，基于某种打分机制得出的绩效结果；沟通记录，个人绩效沟通过程中的记录；过程改进方案，个人过程改进和团队过程改进的思路、目标、措施等的具体描述。

绩效管理是研发团队的老大难问题，与传统行业相比，软件是“软”的，开发过程中的不确定性和变化性确实很难通过量化的方式得到绩效结果。技术管理者尝试基于个人和团队过程改进为绩效管理提供一些思路和工程实践方法。

9.2 向上管理

任何企业或组织，员工和上级管理者之间都需要进行沟通。我们把这种沟通称为向上沟通。但单纯意义上的向上沟通是不够的，还需要对上司进行管理。如同向下管理，向上管理也是技术管理者能够胜任日常工作的一个重要方面。

9.2.1 了解上层管理者

向上管理的第一步同样是要了解上层管理者。但不同于向下管理中对技术人员开展的类型和特点分析，对于上层管理者，我们的首要切入点是他们的管理风格。

业界关于人际风格分析存在一个 DISC 人格测试模型^[42]。DISC 测验是国外企业广泛应用的一种人格测验，用于测查、评估和帮助人们改善其行为方式、人际关系、工作绩效、团队合作、领导风格等。DISC 个性测验由 24 组描述个性特质的形容词构成，每组包含四个形容词，这些形容词是根据支配性（Dominance, D）、影响性（Influence, I）、稳定性（Steadiness, S）和分析性（Compliance, C）四个测量维度以及一些干扰维度来选择的，要求从中选择一个最适合自己和最不适

合自己的形容词。

图 9-12 对 DISC 测试通过四象限模型进行展现。上层管理者自己的性格通过内向和外向两个维度来展现，而关注点则为一般意义上的人和事。我们看到在两两组合之下，可以通过 DISC 模型了解管理者的风格。

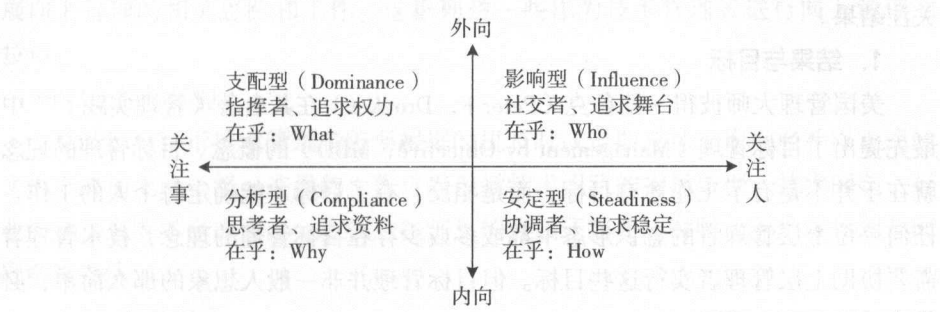


图9-12 DISC测试模型

对于支配型管理者而言，喜欢指挥别人，工作时精力旺盛，说话简单明了，速度较快，甚至一针见血。喜欢冒险，作决定时明快，别人作决定缓慢时，往往会显得不耐烦，甚至给予压力。同时，工作至上，注重成果，并有强烈的时间观念。

影响型管理者，对单调、系统化、逻辑化的事情反感，情绪起伏变化很大，情感很容易流露出来。爱闲聊，有时不切主题，会跑题，说话的声调及音量变化很大，脸上表情丰富，并经常以手势及肢体来强化效果。经常率先提出新计划，喜欢告诉别人事情，作决定明快。

安定性管理者比较注重人情。在解决人际问题时，最能将心比心，设身处地为他人着想。比较在乎别人的评论、反应与想法。面部表情和目光都很友善，态度随和，不带攻击意味，对时间安排有弹性。开会时较沉默，容易配合别人的意见，不会积极主动作决定。

分析型管理者的风格往往可以用准确的、有分析力的、谨慎的、谦恭的、成熟的、有耐心的、严谨的等来形容。善于下定义、分类、获得信息并检验，善于发现事实并保持高标准。他是综合性的问题解决者，有责任心且稳健可靠。

从图 9-12 中我们也可以看到每种类型的管理风格所关注的点也是不一样的，同时他们也具有不一样的情绪化反应。站在技术管理者角度，其面对的往往是对技术领域不甚了解甚至是完全不懂的上层管理者，通过了解他们的风格，接下去的事情就是沟通，并从沟通中寻求反馈与辅导，从而一方面提升自身的影响力，另一方面也要在潜意识里推动上层管理者建立符合团队发展的管理风格。

以上所述的向上管理思想执行起来有较大难度，毕竟权力掌握在上层管理者手上。我们想要获取上层管理者的信任，并实施向上管理的思想，首当其冲需要

把控的一点就是结果导向与目标管理。

9.2.2 结果导向与目标管理

无论何种风格的上层管理者，其最关注的肯定还是战略目标的达成情况，即关注结果。

1. 结果与目标

美国管理大师彼得·德鲁克（Peter F. Drucker）在其名著《管理实践》^[43]中最早提出了目标管理（Management by Objective, MBO）的概念。目标管理的理念就在于并不是有了工作才有目标，而是相反，有了目标才能确定每个人的工作。任何一位上层管理者的意识形态中都或多或少存在目标管理的理念，技术管理者需要协助上层管理者实行这些目标。但目标管理并非一般人想象的那么简单，必须遵循以下四个原则：

（1）目标合理性

目标管理能不能产生理想的效果、取得预期的成效，首先就取决于目标的制定，科学合理的目标是目标管理的前提和基础，脱离了实际的工作目标，轻则影响工作进程和成效，重则使目标管理失去实际意义。站在技术开发团队角度，就可能出现需求范围不明确、开发节奏混乱、开发任务过程繁重、人员流失等一系列问题。可以说，目标的合理性是保障技术开发工作可控的前提条件。

（2）过程监控

目标管理，关键在监控。在目标管理的过程中，随时跟踪每一个目标的进展，发现问题及时协商、及时处理、及时采取正确的补救措施，确保目标运行方向正确、进展顺利。为了满足既定目标，技术管理者应该关注于过程的监控和结果的反馈，从而合理把控上层管理者的期望。

（3）成本控制

目标管理以目标的达成为最终目的，考核评估也是重结果轻过程。这很容易让目标责任人重视目标的实现，轻视成本的核算，特别是当目标运行遇到困难可能影响目标实现时，责任人往往会采取一些应急的手段或方法，这必然导致实现目标的成本不断上升。对于软件开发而言，成本控制相对比较弱化，但作为技术管理者，在过程监控的过程中，也需要对成本保持一定的敏感度，毕竟站在公司角度，技术的投入也并不是不计成本。

（4）考核评估

任何一个目标的达成、项目的完成，都必须有一个严格的考核评估。考核、评估、验收工作必须选择执行力很强的人员进行，必须严格按照目标管理，逐项进行考核并做出结论。关于这点，我们已经在 9.1.4 绩效管理中进行了讨论。

以上目标管理的理念和原则有助于技术管理者在管理技术团队的同时，明确团队目标并获取执行结果。再通过结果与上层管理这之间进行沟通实现向上管理。

2. 向上管理的技巧

如果能很好地帮助上层管理者做好目标管理，技术管理者就有条件和资本开展向上管理的相关思路和工作。这里列举一些作为技术管理者进行向上管理的技巧：

（1）总结信息

总结信息是指技术管理者需要根据时机及时或定期总结工作进展并以正式的文档方式递交给上级。这里的工作进展包括技术团队的指标数据、技术问题、行业关键信息等各个方面。一般在这些信息的背后还需要同时附上下阶段计划及问题解决办法。

（2）沟通约定

与上级沟通的目的在于形成约定，即达成双方对于某个目标的一致认知。通过沟通，需要规范好上级的行为和约定的规范，共同执行。

（3）解决难题

对于上级面临的一些问题，如果能够解决则应该挺身而出，替上级解决困扰他已久的难题。

（4）反馈问题

对于上级所制定的与技术团队相关的明显有问题的重要决策，需要征求团队内部的意见，并给出合理分析建议，反馈给上级。

向上管理的最终目标是为了引导上层管理者的风格，也可以通过培训、分享、个人交流等不同方式，针对技术开发和技术团队管理告诉上级一些这方面的理念和方法，从而让上层管理者更好地了解技术工作的内容。在策略上，可以先和同事商量以争取支持，并提供信息让上级自行改变。在 9.2.1 中介绍的管理风格分析的基础上，适应彼此的个性和风格，迎合他的长处，尽量避免他的短处。更为重要的，有选择地利用它的时间和资源为技术团队服务。

9.3 向外管理

一个公司围绕其战略目标产生了市场、销售、研发等部门。技术管理者作为技术团队的对外接口人，也是处于公司的管理链路中，意味着需要与其他部门进行协作。协作的基本方式就是沟通，但不同的公司、不同的团队多少存在一些政治和文化氛围，技术管理者所面临的外部环境往往与组织架构、环境、工具等因素息息相关。

9.3.1 政治与协商

在一个软件开发团队或公司中，政治（Politics）并不是一个特别敏感、但也不是一个可以置之不理的话题。通俗地讲，政治可以理解为通过与别人协作把事情办成的一种艺术。当然，想让别人与自己协作远非想象中那么简单，关系到双方或多方的利益和冲突，也关系到协作的推动者和被推动者的目标和动机，为了能够让别人和自己站在同一立场和角度去看问题、解决问题，我们同样需要应用方法论，利益、冲突、目标、动机、关系是政治中的关键词。

1. 政治因素

每个角色对政治的理解不尽相同，对于技术管理者而言，承接的是技术负责人的角色和职责，在使用政治这门协作相关的艺术时，环境、时机、原则和策略是所需面对的最基本的几项政治因素，也是我们思考和解决问题的出发点。

（1）政治环境和时机

政治环境和时机上，首先应该把握一个公司的企业文化。企业文化的表现形式有很多，包括对加班的态度、绩效管理、晋升制度、开会方式等。企业文化可以理解作为一种事业环境因素（Enterprise Environmental Factors），即我们只能适应而无法改变。同时，当我们有任何一个想法尝试去推行时，确保该想法与公司的战略方向一致。技术管理者虽然是技术岗位，但对公司高层次的战略方式还是要有一定的敏锐性，从把握领导关注的问题上，可以体会出公司层面的一些想法和做法。最后，政治环境是一个不断经营的过程，尤其体现在人际关系的建立上。

（2）政治原则

政治原则上同时考虑对上 / 对下的入口和出口对于技术管理人员尤为重要，这也意味着需要把握信息透明度。如同在项目管理中项目经理需要考虑项目范围、时间和成本等信息进行有效过滤一样，对上级的向上管理和对下级的向下管理过程中所采用的信息管理策略也需要有所不同，不应该把你所有知道的信息都直白的同步给别人，信息的转换胜于信息的透传。

（3）政治策略

在政治策略上，帮助别人达成目标，很多政治因素也不会因个人努力而改变，学会享受过程而不是目标本身。小事情上可以妥协，但在关键点上，我们争取尽全力出成绩。当出现因人际关系导致的问题时，首先尝试把人际关系问题私有化处理。

2. 协商

很多时候，结果并不是做出来的，而是协商出来的。有些事情看上去很难，但一协商发现并不是那么难，而一点小事如果没有协商可能变成一件大事。

对于协商，换位思考或者说同理心可能是消除协商过程中出现过多扯皮现象

的一大原则。站在对方的立场上思考问题，然后再抛出符合对方立场的言论但能引导到己方利益的观点需要技巧性，这种技巧性的培养需要循序渐进的把握协商过程，而不是对任何问题都直奔主题。

在协商前，最重要的是明确此次协商的目标，包括最低和最高目标。梳理哪些内容是可以或不可以协商的，明确相关关系人，团结一切可以团结的力量。另一方面，充分的准备是协商成功的一大关键，关于这次协商的数据、文档、环境等因素都在考虑范围之内。对于某些预想中比较难以协商的内容，协商前对某些关键关系人或某些关键文件数据等进行私人或内部团队之间的预演可能也是必要的一些环节。

在协商过程中，不要掩盖问题，尤其是对双方都有影响的问题确保在协商过程中明确地提出来，并得出确切的结论。最后，关于协商原则记住一句话，态度要和蔼但立场要坚定。

当协商完成之后，不管协商结果如何，记录必不可少。对于本次协商中重要的决议确保进行文档化和邮件化，如有必要也可以纳入版本控制系统之中进行管理。对决议的执行过程中，明确决议的边界，执行自身相关的任务。但对于技术管理者而言，还要尝试学会委派下面的人进行任务执行。

9.3.2 沟通管理

沟通在互联网软件开发过程中变得越来越重要，可以说，没有沟通，就不可能开发出正确的软件。同时，沟通管理作为项目管理核心知识领域之一，在项目管理和团队协作中的作用也毋庸置疑。沟通管理涉及的范围很广，本节从沟通的重要性 and 模型出发，主要从信息传递和信息维护这两个方面对沟通管理进行阐述。

1. 沟通概述

我们可以通过抽象把沟通过程描述成图 9-13 中的模型，模型中无论是信息编码和解码、发送和接收都会受到多种干扰导致信息的传递出现问题，如何保证信息传递的高效性是技术管理者日常工作中需要进行保障的一个重点。

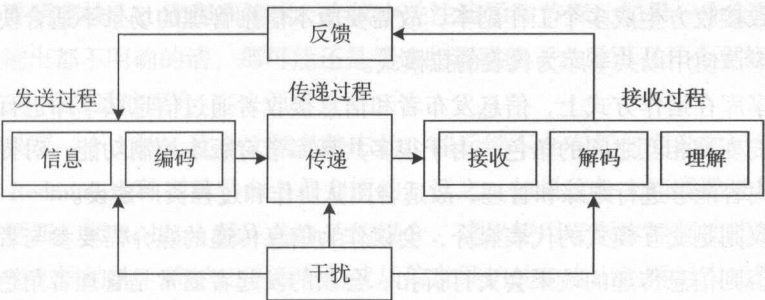


图9-13 沟通模型

有了沟通模型，我们再来回顾另一个重要概念，干系人。干系人模型可以简单抽象为围绕“我”可以引出事情，只有同时满足是我的事情而且与这件事相关这两个条件的人才是我的干系人。干系人只有两种，一种是行动者，即对信息需要采取行动的人；另一种是知情者，即只要知晓信息即可的人。两种不同的干系人决定了所需要传递信息的内容和类型。影响沟通管理的另一个主要方面是组织过程资产，关于过程资产我们已经在 8.4.2 节中做了介绍。而 9.1.3 节中团队管理相关的人员组织结构等事业环境因素同样也会影响沟通管理的具体开展方式，各个组织可能差异较大。可以说，沟通是一个综合性的概念，包含的范围非常广。

2. 信息传递

信息传递的模型可以通过维度（Dimension）、模式（Pattern）和媒介（Medium）三个方面来进一步细化。

信息传递的维度按照不同视角可以有很多类别，一般包括如下几种，每一种参照字面意思即可理解。内部（在项目内）和外部（客户媒体、公众）；正式（报告、备忘录）和非正式（电子邮件、即兴讨论）；垂直（上下级之间）和水平（同级之间）；官方（新闻通讯、年报）和非官方（私下的沟通）；书面和口头。

信息传递的模式只有三种，拉（Pull）模式、推（Push）模式和交互（Interact）模式。拉模式适合受众明确、时效性强，但不适合版本信息管理；推模式则受众面广、平台化管理，适合版本信息管理；而交互模式实时性强、成本高，所以交互议程和节奏是关键。

信息传递的媒介和传递模式紧密相连，这里结合上述传递模式的特点和适用场景分别列举一项最典型的传递媒介。

邮件是推模式的代表媒介，是比较正式且常用的推模式，即我把信息推送给你，至于后续你如何处理就看你的安排，所以适合多方协作但时效性不强，可以在需要明确细节、追踪状态、安排事情等场景使用。但因为推模式的效用只限于本次记录，所以类如对某一个文档不停更新版本并进行通知的场景，每一封邮件都会导致接收方生成多个工作副本，故需要版本信息管理的场景不适合使用推模式，而应该使用以共享库为代表的拉模式。

共享库在运作方式上，信息发布者和信息接收者通过信息共享库进行交互并根据需要变换相互之间的角色。由于很多共享库带有版本控制功能，对提交者以及提交内容能够进行跟踪和管理，故适合团队协作和过程资产建设。

会议则是交互模式的代表媒介，会议作为信息传递的媒介需要参与者做统筹安排，否则信息传递的效果会大打折扣。会议的发起者通常是管理者角色，而接收者可能来自跨职能的各个部门和小组。发起者和接收者之间的意识形态、工作

方式等存在一定差异性，故会议前的准备、会议中的议题和节奏、会后的工作事项落地都会需要成本。相比邮件和共享库，交互模式中的会议是信息传递最需要管理理念渗入的一种媒介。

针对日常工作过程中碰到的信息传递问题，列举若干典型场景。

（1）不必要的干系人

组织内部的邮件通常会以组的形式进行管理，如果这封邮件只是发给某些人，那就不要用邮件组。邮件组是“不必要干系人”的典型应用场景，有些职位的人会加入到很多邮件组中，如果他每天都收到几十封和自己完全没有关系的邮件，那真正需要他采取行动的邮件很可能被遗漏，导致沟通出现问题。另外，在讲技术细节的邮件里抄送公司高层，在讲项目进度的邮件里抄送技术人员，在讲技术方案的邮件里抄送项目人员都是不必要干系人的具体表现形式。

（2）不正确的干系人

在项目启动会上我们会进行该项目的风险分析，如果你把“项目实施人员经验不足”这条风险写到启动会报告中，那很不幸你没有找对信息传递的干系人。“项目实施人员经验不足”确实是一项需要进行内部管理的很重要的风险，但项目启动会面对的是项目的甲方、乙方以及其他供应商，如果你说做这个项目我们的实施人员不行，你让其他方的人怎么想呢？

（3）不合适的维度

典型例子有通过 QQ 传递重要信息；口头通知项目决定；项目数据非可视化沟通；缺少内部 / 外部信息过滤等。

（4）不合适的模式

如果你想和团队成员分享一个很好用的小工具，那建议你不要用邮件去传递信息，因为邮件可能会被删除和遗忘，这种场景下运用拉模式通过 SVN 或 FTP 等共享库进行信息传递往往是更好的选择。

（5）不完备的模式

主要是对会议而言，上面也提到会议需要进行统筹安排方能发挥其效用。会议前需要明确输入、议程和输出；会议中关注演示和节奏。如果一个会议连基本的输入输出都不明确的话，那可能还是等这些都明确了之后再召开会议更好。

（6）不合适的媒介

如果写一个文档，这个文档是静态的，即后续不会有任何变动和更新，那把它放到 Redmine 这种工具平台上是合适的。反之，如果这份文档需要进行版本的演进和更新，那 Redmine 就不是合适的媒介，建议使用带版本控制功能的共享库进行这些文档的统一维护。下面我们就从信息维护的角度出发再对沟通管理进行进一步分析。

3. 信息维护

信息维护是一项涉及知识库、过程资产、环境和交流等元素的整合过程，该过程包括信息保存、转移和转化的成本，由于这种成本比较隐性，很多时候我们或多或少不想投入这种成本，导致信息维护的完整性和时效性上出现问题。

同时，信息维护一般会借助于某些工具以达到自动化或半自动化的效果。首先是版本控制工具，如果信息需要分版本、需要定期 / 不定期维护、需要团队多人协作，那版本控制工具是必需的，主流的版本控制工具包括 SVN、GIT；问题跟踪工具，如果信息的特点是随项目 / 产品开发进程不断需要范围变更、问题抛出和解决、多方干系人参与，那采用一套完备的问题跟踪工具会事半功倍，主流的包括 Redmine、Jira、Quality Center；静态资源工具，如果信息只是一些静态资源，不涉及变动，那就用 FTP；知识共享工具，如果信息属于知识管理范畴，那采用一个知识共享工具能帮助团队解决很多耗费精力但成效低下的信息共享和维护需求，主流的包括以各种 Note 为后缀的工具。

信息传递和维护是沟通管理中的两个重要方面，日常工作无论是研发、项目、产品以及高层的管理工作都需要信息的高效传递和维护。各个组织有各自的特点和文化氛围，技术管理者在从事沟通管理时需要通过探索形成统一、合适的方法论和工作模式，并进行持续改进。

9.4 自我管理

自我管理也是一种管理，而且执行起来可能比上面几个管理都要困难。作为团队中的一个个体，自身存在的一些思路、习惯、工程实践以及可能让其他人感到反感的问题行为都是自我管理的对象。但个体自身通常很难意识到这些问题，尤其对于技术管理者而言。你的下属通常不会直接向你提出这些问题，就算他们具备 9.2 节中介绍的向上管理的方式方法，通常也很少会有人针对性地进行对你进行向上管理，因为他们是技术开发人员，而不是像你一样的技术管理者。

要有效地实现自我管理，实际也是一个过程改进的过程。首先需要对自己的习惯、实践和行为进行评估，然后明确需要改进的切入点，最后执行你自己制定的改进计划。要评估的方向有很多，我们认为最重要的是两个方面，即个人性格和处理事情上的方式方法。

9.4.1 个人风格

我们在 9.2.1 节中提到了 DISC 风格模型来判断上层管理者的管理风格，而对于你的下属而言，你就是他们的上层管理者，所以这里的个人风格我们也可以沿

用 DISC 模型来进行进一步展开。从这点上讲，技术管理者做管理风格分析，并不是为了了解对方，掌控别人，而是为了更好地了解自己，从而确定如何和别人相处的个人风格。

所谓个人风格实际上有几层含义，第一种是个人的内在行为模式，即一个人天生的、固有的行为模式，代表最自然和真实的动机；第二种是外在行为模式，是个人基于自身对环境的判断和认知，认为自己在特定环境下理应呈现的理想行为模式，这种模式通常代表个人试图在工作中采用的行为类型；第三种称为认知行为模式，即每个人自己都有一种特定的认知，继而产生一种特定的行为模式，这种行为模式是个人基于过往经验和环境的一种结合，通常所说的“自我”就是指这种模式。

显然以上三种个人行为模式中，相对比较稳定且被周围人所熟知的是第三种模式，我们通过 DISC 模型进行分析的也正是这种模式。图 9-14 是基于 DISC 模型自我分析图示例，我们结合 9.2.1 中关于四种类型的表现形式可以做成自己的评估，然后我们对每种评估结果做一一展开。

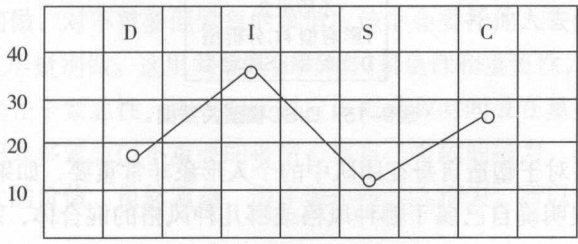


图9-14 DISC模型自我分析图示例

如果你的评估类型是支配型风格，那么你的主要问题就在于不容易看到别人的需求而只看到自己的需求。注意不要演变成乾纲独断的风格，这样容易让别人感到压力，相处起来就比较累。在自我规划上，支配型管理者可以学会减轻对别人的压力，学会放松。多尝试接受别人的意见，并尝试变得耐心和低调。

影响型风格的主要问题在于往往以自己为中心，独占主体，喜欢打断别人的谈话，不注意记忆，容易变化无常。喜欢做的事情有很多，但缺乏持之以恒的毅力。如果你是一位影响型管理者，在自我规划上需要控制自己的表现欲望，要脚踏实地把一件事情从头到尾做完整。

安定性风格不容易兴奋，不喜欢变化、也不善于做决定。对压力往往采用回避策略，不愿意承担责任。要注意安定性管理者有时会表现得毫无主见，随意在自我规划上增加一些新鲜事物和思想。需要明确工作的责任，有意识地去接收别人的监督，并多与其他人沟通表达自身的想法。

分析型风格的主要问题在于执行力比较弱，太容易思考而容易产生悲观情绪。同时容易受外部环境影响，在变化比较快的节奏下比较敏感，容易产生情绪低落。分析型管理者的自我规划应该不要把时间都用来规划而不去付诸实施，也不要用自己的高标准去要求周围的同事，控制自己对外界环境的反应，不要太敏感。

我们可以用“做、说、看、想”这四个词分别概括以上四种风格。在四种类型中，实际上软件开发人员比较接近的应该是分析型风格，而分析型风格恰恰是最不适合做管理的一种风格。最适合成为技术管理者的一般认为是支配型风格。但显然不是每一个人都能简单的一刀切成一个风格，更多的情况下，个人表现出来的是一种混合型的组合类型。

个人风格的组合类型同样也有几种不同的表现形式，包括自然组合、互补组合和矛盾组合，图 9-15 展现了这几种组合关系。



图9-15 DISC模型关系图

不同的风格对于塑造自身在团队中的个人形象非常重要，如果通过以上分析，你内心还不是很明确自己属于哪种风格或哪几种风格的混合体，那么可以参考表 9-2 中所列举的一些示例，用于帮助判断自身的个人风格。

表9-2 四种风格举例

对比维度	支配型	影响型	安定性	分析型
第一印象	做事快	交朋友快	动作慢对人细心	做事非常仔细
对于压力	争	闪	忍	闭
遭遇逆境	攻击挑衅	找理由合理化	逆来顺受	退缩不前
人格特质	有行动力	有魅力	亲和力	有智慧
不喜欢	不求效率	墨守成规	出其不意	朝令夕改
事业目标	财富权势	名声影响力	稳定认同	专业权威

个人风格评估不是终点而是起点，本节提供的基于 DISC 模型的评估结果为技术管理者想要通过个人努力而希望达成的目标风格提供方法和依据。

9.4.2 处理事情

作为技术管理者，最重要的工作并不是冲到一线做各种技术研发，而是要处

理各种技术以及一些非技术相关的各项事宜。所以处理事情是自我管理中除了个人风格外的一个主要主题。处理事情需要做到对时间的合理利用以及对事情的管理和跟踪。

时间四象限模型是美国的管理学家科维提出的一个时间管理的理论，把工作按照重要和紧急两个不同的维度进行了划分，可以分为四个象限，见图 9-16。

对技术管理者而言，每个象限的典型事情如下：既重要又紧急，如用户投诉、线上服务问题、广告运营紧急需求等；重要但不紧急，如建立人际关系、人员培训、制订工作流程等；紧急但不重要，如面试安排、部门会议等；既不紧急也不重要，如邮件处理、写博客等。

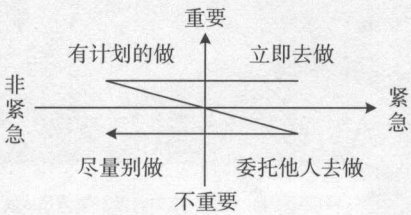


图9-16 时间管理四象限模型

技术管理者作为整个技术团队的统一入口和出口，对每件事情所持有的态度上应该有基本的原则，对重要且紧急的事情应该立刻去做，对重要但不紧急的事情可以有计划的做，对不重要但紧急的事情应该学会委托他人去做，而对不重要不紧急的事情就尽量别做。这里关键是如何比较紧急性和重要性，在多数场景下，重要性实际上优先于紧急性，即我们平时应该把多数时间放在重要但不紧急的事情上，而不要把太多紧急但不重要的事情占据自己有限的精力，因为不重要的事情并不能产生太大的价值，虽然紧急，但不做也没什么损失，所以还是要先管重要的事情。

以上通过时间管理的理念对做不做事情做了分析，即时间管理的关键是保持一份每天需要完成的待办事项清单，然而跟踪这些待办事项同样重要。正如你向上级汇报一样，需要时刻关注工作完成情况。一般需要维护日常任务清单，这份清单可以通过一些电子化工具在你的工作电脑或手机移动端进行提醒。

9.5 本章小结

组织管理从软技能角度对技术管理者提出了转型要求。组织管理涉及一个团队运作和管理的方方面面，本章从向下管理、向上管理、向外管理和自我管理四个角度出发来梳理技术管理者的日常组织管理工作。

向下管理的对象是团队中的技术人员，涉及领导、激励、团队日常管理和绩效管理等内容；向上管理则需要了解上层管理者，偏向于结果导向和目标管理；向外管理的基本思路是通过沟通管理达到政治和协商上的团队目标；而自我管理偏重于个人风格的建立以及对如何有效的处理事情的把握。

非卖品！！严禁（售卖和上传互联网平台）！！违者责任自负！！

成功转型篇

向技术管理者转型

软件开发人员跨越行业、技术、管理的转型思维与实践

本篇是全书的总结篇，在前面各篇的基础上，讨论想要成为合格的技术管理者所应具备的意识形态上的转变。管理者首先应该具有特定的思维模式并通过引入变化去面对技术管理上的各种问题，促进团队研发文化的建立和发展。

为了成为一名合格的技术管理人员，需要明确工作的层次以及自身在组织架构中所处的定位。技术管理者一般处于公司的规划层，理解战略、定义产品、规划方案、技术实现、过程管理和团队事务构成了日常工作内容的主体。

本篇共有一章，总结全书。

10 成为一名合格的技术管理者

技术管理者是一个综合性的角色，需要熟练掌握业务和技术体系，同时具备良好的组织管理能力。在本书 1.2.1 节中介绍了技术管理者的主要职责和所开展的活动，无论这些活动是偏向技术还是偏向管理，技术管理者都处于一个特定的环境中，也都需要去改变这个环境，从而使这个环境能够适应组织发展战略和目标。

我们围绕技术管理的角度进行展开可以得到图 10-1 中的技术管理者能力模型。在前面的章节中，我们分别介绍了业务产品分析、技术架构和组织管理方面的内容，这些内容构成了本书的三个转型维度。而人员、工具、流程等因素构成了我们所需创造的研发文化，这些因素都与日常的技术开发工作息息相关，也是技术管理者需要关注和打造的对象。

在图 10-1 中，我们还看到两个非常重要的概念，即意识形态和环境变革。技术管理者在技术和管理两个方面都可能成为一个团队中变革的引入者。软件行业

发展日新月异，尤其是互联网行业下的软件开发往往伴随着技术和业务上的剧烈变化，业务创新需要变化，技术重构也需要变化，引入变化并作为主要的推动力去落实变化，也是提升技术管理者在特定环境中影响力的一个方面。而变革的引入首先需要的是意识形态的确立和转变。

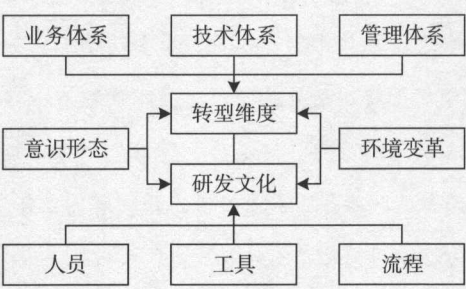


图10-1 技术管理者的能力模型

10.1 技术管理与意识形态

意识形态（Ideology）决定高度，对于管理者而言尤其如此。所谓意识形态，可以理解为对事物的理解和认知，人的意识形态受思维能力、环境、教育、价值取向等因素影响。不同的意识形态，对同一种事物的理解、认知也不同。在本书中提出意识形态的原因在于技术管理者通常处于一个团队的中上层，其自身的意

识形态会对整个团队造成巨大影响，管理者的意识形态的高度很大程度上决定了团队的发展高度。

如何把抽象的意识形态和具体的研发过程结合起来，存在一个闭环管理模型。对于软件开发而言，管理者首先应该具有特定的思维模式去面对技术管理上的各种问题。针对这些问题，解决的思路除了前面几章介绍的业务、技术和管理维度之外，往往需要引入变化。而引入变化的过程通常不是一步到位，需要一个过程，这个过程同样会不断促进研发文化的建立和发展。然后研发文化的成熟反过来又会推进管理者形成新的思路模式。

10.1.1 思维模式

开发人员，尤其是年轻开发人员的一大特征就是情绪化思维，对碰到的问题倾向于使用主观意识去寻找方法，同时又有一些顽固，钻牛角尖的场景并不少见；而技术管理者通常具备全面的思考和分析模式，倾向于使用换位思考从问题的内因、外因出发，找到团队内部和外部能够解决问题的资源，确保问题得以高效解决。这就是不同人具有不同思维模式的示例。从技术到管理的转型，我们认为思维转变是关键。那么什么是思维模式？

1. 关于思维模式

如果把人的思维视为物质的一种特殊的运动状态，借用物理学惯性定律，对思维惯性作这样的解释：如果没有外界因素（外力）的作用，人的思维总是力图保持原有的思维（运动）状态不变，这一特性就是思维惯性。人的思维状态是动态的，但其所形成的思维模式，相对来讲具有一定的稳定性，这种稳定性形成人的思维惯性。

一般认为，人的最基本的思维模式有三种普遍的表现形式，即形象思维、抽象思维、灵感思维。形象思维是借助于具体形象来展开的思维过程，比较容易理解。而抽象思维是运用概念、判断、推理等来反映现实的思维过程，亦称逻辑思维。灵感思维与潜意识密切相关，表现在处理事情的过程中不知不觉突然而迅速发生的特殊思维形式。

开展一项工作或解决一个问题，至少是两种思维并用，即抽象思维和形象思维。例如对于软件开发而言，抽象思维就是最重要的一种思维形式，本书第4章中介绍的技术理论实际上就是对现实开发过程中所重复出现的问题以及解决方法在技术体系上的体现。又如本书第5章中的系统架构设计中提到的多视角、多视图对架构进行分析的方法，应用抽象、扩展、复用、自治等理念，这些点都自包含在架构设计过程中。

而对于管理而言，有时候还需要灵感思维。在行业分析和产品设计过程中我

们经常会使用思维导图。而在进行敏捷回顾过程中，我们也会使用头脑风暴等手段来集思广益。思维导图和头脑风暴就是使用灵感思维的典型例子。管理上的灵感思维有时候比抽象思维更为重要，毕竟管理在一定程度上就是把管理者自身的思路 and 想法通过语言、文字等方式传达给别人的过程，管理者本身并不一定亲力亲为，但需要告诉别人做事的方式方法。灵感思维并不容易获得，需要培养和练习。关于灵感思维的进一步讨论已经超出了本书的内容范围，读者可以参考相关著作。

2. 对比管理型思维与技术型思维

鉴于管理人员与技术人员文化认同和实践感悟上的差异，他们的思维模式呈现出不同的特点：

(1) 管理型思维侧重于整体，而技术型思维侧重于局部

管理人员总是把全局作为考虑问题的出发点和归宿。处理工作时讲究统筹兼顾、全面安排，照顾到各个局部，使各个局部有机地协调起来。注意研究事物的结构和过程，通过优化结构和过程提高全局的整体功能。把注意力的重心放在对全局有决定性意义的问题和事项上。

技术人员考虑问题往往关注局部和事物本身，并深入思考从而得出结论，他们从一个点可以深入挖掘，潜心了解其内在规律、运作模式等。至于他的工作在全局中处于什么位置，与其他人的工作是什么关系，将产生哪些关联影响，有时候并不太关心。

(2) 管理型思维多凭直觉顿悟，而技术性思维依靠逻辑理性

管理人员注重直观经验，习惯于直觉体悟，擅长思辨，基于过去长期积累的经验和认知，靠洞察力、潜意识进行思维，常撇开细节枝节和过程，直指问题的本质和核心。比如他们对事物的把握，习惯从性质上加以判断，注重“定调”。

而技术人员擅长的是建立在经验与逻辑基础之上的理性思维，注重对事物分门别类，重视定量分析和精确计算。他们做人中规中矩，棱角分明，率性而为；做事讲究章法，遵从规则，按部就班，程式化作业。

(3) 管理型思维是先总后分，技术型思维是先后分总

管理人员的思维过程表现为从宏观到微观、从大到小、从整体到局部。他们看问题和处理问题先是建立一个整体概念，然后再做分解。比如，布置工作上他们首先强调的是总体目标，然后再考虑分工的安排、实施的步骤、时限与要求、协同与配合。

技术人员的思维过程表现为从微观到宏观、从小到大、从局部到整体。比如，在实现某个具体功能，技术人员倾向于先搞定较为复杂和困难的领域模型部分，然后再考虑如何设计对外的访问接口和用户界面。

从管理型思维和技术型思维的对比中我们发现不同的思维模式实实在在影响着不同人的工作方式。但是,我们也可以认为人的思维惯性不是一成不变的,只是具有相对的稳定性而已。一旦思维惯性被打破,经过一段时间又会形成新的思维惯性。人的思维的发展,需要不断打破思维原有的稳定性和思维惯性,这样才能形成新的思维的稳定状态。而外界因素的作用要足够大到能克服思维惯性,人的思维才能摆脱原有思维模式的束缚。

3. 技术管理与组织思维模式

要想成为一名合格的技术管理者,技术开发人员需要摆脱原有的思维惯性,通过变化形成一种新的、适合技术管理的思维模式。除去上文中讲到的最基本的形象思维、抽象思维、灵感思维之外,技术管理者转型过程中最具针对性的需要转变的思维模式就是组织思维。

组织思维就是由组织中人员的个体思维,尤其是组织的领导者或组织核心层人物的个体思维组合构成,其中组织领导者或核心层人物的思维方式对组织思维影响巨大。组织思维同样具有惯性,组织思维惯性是组织经过一段时间、一些过程所形成的,并被组织成员所接受的思维模式、思维方法,它是组织思维的沉淀。显然,组织结构的大小及组织结构的紧密程度,对组织思维惯性、组织行为惯性有着重要的影响;组织结构越大,组织结构越紧密,组织的思维惯性、行为惯性就越大。反之,组织结构越小,组织结构越松散,组织的思维惯性、行为惯性就越小。

改变组织行为惯性的关键在于克服组织思维惯性。克服组织思维惯性的关键,又在于组织的领导者或组织的核心层人物。因此,要克服组织行为惯性,首先要克服组织领导者或组织核心层人物的思维惯性。技术管理者就是组织的领导者和核心人物,这就是为什么技术管理者在转型过程中需要特别注意组织思维模式的原因所在。接下来,我们来讨论如何进行组织思维模式的转变。

10.1.2 引入变化

不断创新及不断适应社会发展的需要,正确识别组织需求及确定组织目标,充分发挥组织成员的参与性,充分调动组织成员的积极性、创造性,这是克服组织思维惯性、组织行为惯性的的重要途径。为此,我们需要通过引入变化来达到技术管理者自身思维模式以及组织级别思维模式的转变。

1. 对引入变化的理解

过程改进是研发管理的本质性工作,如果过程要改进通常意味着我们要引入变化,尤其对当前研发管理尚不规范和完善的团队而言,引入变化是必须走的一步。通常管理层都能认识到引入变化的重要性和必要性,但很多中小型企业普遍

重技术而轻管理，很少会有团队把引入变化作为一项专职工作来做，也就很少有流程和实践来指导相关工作。本节的初衷就是希望把引入变化作为团队中一项工作内容来管理，为过程改进的顺利开展提供可行的操作模式和实践。

引入变化的流程如图 10-2 所示，每个步骤都需要专职角色进行负责和把控。其中发现问题是指针对团队中的不良现状，抽象为需要引入变化进行解决和改善的问题点，然后通过收集、分析数据对问题点进行剖析并找到切入点，应用模式上，组合应用下文中提到的各种模式进行变化引入，最后是跟踪与回顾，根据所引入变化的效果和团队的反馈进行总结和回顾。对如何发现问题、如何找到切入点以及如何开展回顾需要因地制宜，技术管理者可以直接借鉴的是“应用模式”部分。

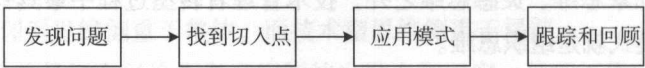


图10-2 引入变化的流程

对团队引入变化，很多人存在一些误解，其中最大的误解就是认为新想法一旦引入就意味着已经成功。团队管理人员想了很多办法、做了很多工作找到了问题的切入点，跟各个利益方讨论之后终于引入了大家都赞同的变化，然后就期望这个变化能按照自己想的那样发挥效果，这是不对的。新想法一旦引入，我们还要做很多事情，很多变化引入的失败并不在起点，而是在过程的持续性上。如果一个团队的自组织性较差，没有流程进行约束，也没有专人进行跟踪和协调，变化的结果通常不会令人满意。

2. 引入变化的思路 and 模式

引入变化在思路上第一点是自下而上，全员参与。很多人认为变化应该有管理层发起，然后下面的人配合执行，这种观点没有错，而且有时候也是必要的。但反过来，自下而上进行变革，让所有人都能参与进来往往是一种更优解。举身边的一个例子，开始时研发团队普遍没有过程资产管理这一概念导致开发交接上出现很多问题，后来部门里的一个小组开始推行基于 OneNote 的研发知识管理，把项目线、产品线和技术线上的很多内容都放到 OneNote 进行共享，效果非常好。周围的小组看到这个情况，也开始推动研发知识的管理，后来扩展到整个部门，再后来整个组织的研发人员都会登录到 OneNote 知识管理平台上做分享和交流，公司管理层知道后也非常赞赏。这个就是自下而上，全员参与的典型实例。

关于团队引入变化的模式，业界也有一些主流方法^[44]，包括专职负责人、关系网、团队培训、电子平台、回顾时间、自身经历分享、不妨一试、适可而止、按部就班、个人沟通、合适时机、小有成绩、学习小组和试行等。这些模式我们都会展开探讨，但使用这些模式时需要把握几个要点。在使用场景上，我们认为

上下文环境是引入变化最重要的考虑因素，如果团队现状和公司战略与新想法不匹配，显然这个新想法就不适合引入。对时机而言，可能研发团队一直都很忙，但在推行一些新想法的时候发现，如果这些新想法确实能为团队带来工作效率的提升或客户满意度的提高，通常都会很受欢迎。时间确实都是挤出来的，关键是要有人去主导。而对象上，针对某一个新想法都要找一个合适的专职负责人，这个专职负责人可能通常都是团队管理人员，但最好多培养团队普通成员。对团队而言，重点关注对变化漠不关心或不大参与新想法实施的成员。

10.1.3 研发文化

对于研发团队而言，引入变化的目的是为了打破个人和组织的思维惯性，从而形成新的研发文化。

1. 研发文化的维度和切入点

研发文化的建立需要因地制宜，一般取决于两个重要条件，即团队所处的不同发展时期以及团队所承担的产品开发特点。同时，技术开发本身具有的一些特点也是研发文化的一部分。

(1) 团队生命周期和研发文化

根据 Bruce Tuckman 的团队发展阶段模型^[45]，一个团队的发展一般需要经历 5 个阶段，即组建阶段（Forming）、震荡阶段（Storming）、规范阶段（Norming）、执行阶段（Performing）和休整阶段（Adjourning），并且团队在成长、迎接挑战、处理问题、发现方案、规划、处置结果等一系列经历过程中必然要经过上述五个阶段。团队发展的不同时期其研发文化自然也有不同的表现。从图 10-3 中，我们可以看出各个阶段的团队精神和工作绩效会有明显的变化。

在组建阶段，由于某个产品或项目的创立促使大家坐在一起工作，团队整体是一种积极向上的愿望，急于实现某个计划。但团队成员并不一定非常明确自身以及其他成员的职责和角色，个人对工作本身和其他成员的关系持有一定的疑虑态度。这阶段的研发文化实际上是非常不错的，大家都想做很多事情，但结果恰恰是很难取得实质性的工作进展。

在震荡阶段，团队目标和分工开始明确，团队成员开始运用自身的技能执行分配到的任务，

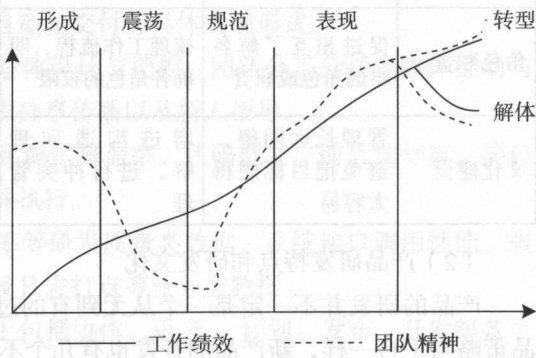


图10-3 团队发展的四个阶段

整体工作得到推进。但随着工作的开展，成员对团队领导者的风格以及自身的角色和职责会产生更多的疑问。某个流程或制度被制定并付诸实施时，很多人会怀疑这类流程或制度的实用性和必要性。这就会导致各种冲突的不断产生，团队气氛开始紧张，成员之间会形成抵制甚至对立的情绪。这阶段的研发文化应该说是处于整个团队成长生命周期中的最低谷。

在规范阶段，团队成员之间的关系已经明确，大部分冲突和矛盾也得以解决，成员明确并接受了各自的角色。同时，团队的运作规则得以改进和规范化，一些关于研发方法和过程也慢慢由团队成员自身进行决策。通过有效沟通和协调，成员之间建立起信任感和责任感。这阶段的研发文化得到快速提升。

在执行阶段，成员积极工作，集中精力实现团队目标。团队有集体荣誉感和凝聚力，团队成员之间的协作机制高效而透明。这阶段的研发文化在整个生命周期中发展到顶峰。

在休整阶段，一般研发文化会随着团队使命的终结或衰落而趋于平淡。研发文化更多表现为一种失落感，关于团队未来的不确定性开始回升。

站在团队生命周期的角度来看研发文化，技术管理者需要根据不同阶段的特点合理利用业务、技术和管理三个维度来确保在每个阶段都能使团队正常发展。表 10-1 列出了在组建、震荡、规范和执行这四个阶段中应该采用的技术管理措施。

表10-1 团队发展阶段与技术管理措施

	形成阶段	震荡阶段	规范阶段	表现阶段
项目目标	明确项目目标，建立共同的项目愿景	分解为具体目标，并进行适当的授权	鼓励项目成员制定日常工作目标与计划	关注结果，制定挑战性目标
团队规则	明确团队规则，进行团队培训	解释规则，并维护规则的有效性	改进规则，并进一步规范	促进成员自觉维护规则
角色职责	促进相互了解各自的角色或职责	梳理工作流程，明确各角色的权限	根据实际情况，对职责及关系做出调整	淡化职责边界，倡导相互补位
文化建设	管理长远期望，避免把目标想得大容易	增进沟通和理解，进行冲突管理	倡导和加强团队精神	鼓励坦诚，强化团队凝聚力

(2) 产品研发特点和研发文化

产品的研发并不一定是一个从无到有的过程，正如我们在 3.2 节中提到的产品策略与平台一样，新产品的开发也有几个不同的表现形式，而不同的产品研发类型也决定了产品的开发团队类型。表 10-2 列举了三种比较典型的新产品开发

类型以及所对应的团队特征。

表10-2 新产品开发类型

产品开发类型	团队类型	团队特点	管理要素
新平台产品开发	独立的专项团队	独立于日常开发	保持团队高度自治
完整现有产品线	跨部门团队	开发工作与其他日常工作并行	加强部门协调机制
产品技术改进	技术改进团队	涉及范围较小，组建方式灵活	把握项目运行的时机

研发团队文化是企业整体文化的组成部分，因此它具有企业文化的共性。但根据不同的产品研发特征，又有它的独特性和自身要求。例如，开发活动的创新性就要求鼓励团队成员原创性的工作，但鼓励创新也必须建立在团队工作的基础上；开发活动的协同性要求鼓励研发团队成員随时随地的交流，要有相应的沟通机制；开发活动的不确定性，即风险性要求研发团队重视工作细节和团队成员不同的意见；开发活动的时限性则要求团队成员要有强烈的时间观念和责任意识。

以上这些根据产品研发特点而对团队成员提出的要求构成了团队研发文化的基本基调。技术管理者首先需要判断当前团队所从事的产品研发类型，并根据创新性、协同性、风险性和时限性等在该产品研发过程中占有的比重来打造合适的团队研发文化。

(3) 技术开发特点和研发文化

讨论完团队、产品与研发文化之间的关系，我们再来看一下技术开发本身的特点以及与研发文化之间的关系。技术开发上的各种特点与技术人员息息相关，而且在日常开发过程中也不知不觉就形成了一些特定的文化体系。这些文化体系在大多数技术团队中具有代表性，包括工具文化、过程文化、代码文化、数据文化和工程实践文化。

对于工具而言，技术管理者的职责涉及工具、模板、规范的制定和推广落地应用，包括配置管理、持续集成、自动化交付等具体表现形式。

对于过程而言，涉及实现方案总体设计与把控；团队分工分任务整体推动与协调；团队项目复盘总结、经验知识分享传播以及新人指导。

对于代码而言，代码评审、单元测试覆盖率、代码框架、前后端分离、接口定义通常需要一定的手段形成规范并执行。

对于数据而言，包括系统缺陷率等研发质量类数据，系统接口调用性能、服务器运行状态等运行时状态数据以及日志打点等业务类数据。

对于工程实践而言，研发过程上包括协作、思考、计划、发布、开发等各个维度的实践方法。典型的站立会议、迭代思想、客户测试、重构、技术预演等都

属于常见的工程实践。

团队生命周期特征、产品研发特征和技术开发特征构成了研发文化的切入点，这三者之间形成一个整体。一般而言，产品研发过程驱动团队的生命周期，而产品研发特征又会对技术开发特征有较大影响。

2. 塑造研发文化

一个好的研发文化需要一个良好的工作环境，充满尊重和公平的氛围容易培养并促进承诺和激励的环境，并让员工保持最高效率。同时，关于产品、项目和可交付物有一定的度量标准，用以衡量团队的努力程度并改进结果。那么如何建立这样的研发文化？我们先来讨论一下另外一个与研发文化有直接关系的概念，即公司文化。

所有公司都有自身的文化，不管公司或大或小，或传统或互联网化。如果公司文化积极而有力，那么公司文化相当于就是一个良好的平台，技术管理者可以利用它为自己的团队创造有有利的环境。而如果公司文化偏向消极或与团队的发展现状不符，那么就需要对它进行隔离，从而为团队提供一个独立环境。

对于技术管理者而言，关于公司文化最重要的一个关注点在于要明确技术在你的公司中的地位 and 重要性如何？有的公司将技术看作是盈利的来源和价值的核心，而有的公司则认为需要不断削减技术的成本，甚至有些公司会把技术团队完成看成是一种辅助性的支撑部门。这些观点也会因为行业的发展以及公司产品的演进而随之变化。例如对于电商类创业公司而言，初期普遍技术驱动或产品愿景驱动，而当公司达到某一阶段时，可能会把重心转移到客服和用户体验上。我们可以通过分析得出哪种力量在推动着公司的发展。有些公司是技术驱动型，有些则由市场和销售驱动。很明显，这种推动力会很大程度上决定技术在一个公司的地位和重要性。

公司文化属于事业环境因素，我们一般很难改变，尤其对于大公司而言更是如此。技术管理者处于一个特定的公司文化中，真正可以推动研发文化建设的主要手段体现在精神层面和制度层面两个方面。

（1）精神层面

充分认识到开发工作的重要性是至关重要的，但并不是每个技术管理者都会意识到这一点。国内的大多数企业，传统行业偏向销售市场驱动，而互联网行业则由运营和产品来推动整个研发工作。技术团队普遍处于整个研发流程的末端，地位偏低没有话语权。这个时候，技术管理者需要能够站在开发人员的立场看问题，竭力支持团队开发工作的开展。

在精神层面塑造研发团队文化最好的方法是技术管理者的率先垂范和团结一致，而不是期待开发人员的自觉性。由于开发活动的特性，要鼓励团队成员的创新

活动。鼓励团队成员的良好沟通和共同协作，即团队精神。要求团队重视工作细节和不同的意见；要强调团队成员强烈的时间观念和责任意识。在精神层面上，我们可以通过引入一定的变化来改善当前的研发文化。这方面引入变化的模式包括：

- 寻求帮助。

这是看上去很容易但做起来并没有像看上去那么容易的一个模式，尤其对研发人员而言。很多研发人员包括处在研发管理岗位的 Leader 们，都很善于处理技术上的问题，但沟通和协作上总感觉有所欠缺。在组织中引入一个新想法在工作量上都不是一件小事情，尤其是对于那些刚工作的新人而言缺乏触类旁通的思路和技巧，想让所有人都高效参与到引入变化的过程中来难度很大，这时候就需要找同事和资源协助你一起努力。这个看似可有可无的模式在实践过程中确实会起到一定作用，至少对于技术管理者这一角色而言是这样。

- 关系网

关系网模式和“自下而上，全员参与”这一思路有关，有时候变化想要获得大家的认可，需要有人来帮你造势和宣传。如果一个人在团队和组织中人缘很好，那就建议由这个人去推动一项新想法。当然，作为团队的管理者自然会有很多机会接触到其他团队的做法和成果，把别的团队的新想法吸收到自己团队中或者把自己团队中好的做法推广到其他团队也是一项管理内容，这个时候合理利用关系网模式会起到潜移默化的作用。

- 自身经历分享

如果某个新想法你有类似的经验，那么恭喜你，这个专职负责人非你莫属。鼓励一些成功尝试新想法的团队成员分享他们的使用经历，有助于所有团队成员看到新想法的价值。这个时候，使用相对比较正式的团队培训是可以的，但更加鼓励大家在非正式的场合下以互助的方式分享对新想法的体会和心得。对技术人员而言，有时候并不是不愿意吸收新事物，而是苦于大家知识水平的局限而无从下手，所以如果有人愿意分享经验，作为管理者就要帮助创造这样的机会。

- 个人沟通

如果你想让别人听你的，请你保持与别人的沟通，因为要想说服一个人接受新想法，一定要让他了解这个新想法对他个人的好处。如果你只是在团队培训时提到了这个新想法，大家都觉得它不错，然后你就指望所有人都能去实现这个新想法是不现实的。尤其是研发团队中往往存在不善交际的成员，他们在开会等群体活动中并不一定会把自己的想法说出来，但不代表他们没有想法，如果不去找他们沟通，他们就不会把这些话说出来。研发团队中有时候出现的各种问题就是因为很多人没有把该说的话说出来所导致，这时候技术管理者可以灵活使用“个人沟通”这一模式确保团队中所有人的想法都公开和透明。

· 学习小组

很多组织会设立类似的学习小组，有些模型也主张要有一个小组来统领团队的过程改进工作，如 CMMI 的 SEPG。我们这里的“学习小组”不强调类似正式的组织性架构，而是指一帮有共同兴趣的团队成員可以组成一个小组，让他们能够有机会持续的做一些学习和交流。当然我们认为管理工作不能少，所以定期/不定期的组织一下小组会议还是重要的，最好这个组织会议的人不是来自学习小组中的成员。上面提到的“自身经历分享”等模式都可以和“学习小组”配合使用。

(2) 制度层面

制度层面的重点在于建立适合的产品开发管理机制和研发团队绩效考核机制，并且依据考核结果做到奖罚分明。建设开发团队的工作机制，其目的在于沟通信息、明确责任、协调进度。工作机制可以分为两种，即正式机制和非正式机制。正式机制多体现为组织级别的规章制度，非正式机制则是不同部门的开发人员之间的随机交流。对于那些非正式的团队机制是不好用制度规定下来的，很大程度上受到团队文化的制约。

在制度层面，塑造团队研发文化的主要方式就是建立各种适合自身团队发展的非正式团队机制。这些非正式团队机制的建立每个团队通常都有自己的方法，而通用的方法我们已经在第 8 章中介绍，此处不再展开。但也存在一些引入变化的模式可以用来改善当前的研发文化。在制度层面上引入变化的模式包括：

· 专职负责人

从流程执行、阶段评审、高效决策等角度出发，我们都深信有或没有专职负责人是不一样的，引入变化也是一样。对引入变化而言，对该变比持支持态度并有强烈兴趣的人无疑是专职负责人的候选，从这个角度讲，不同的问题和切入点、不同模式的专职负责人都可能不一样，所以专职负责人通常是流动和不固定的，鼓励团队中的不同成员成为这个专职负责人也是一项最佳实践。专职负责人的作用和职责就是确保在团队内部引进新想法过程的有效性，努力把新想法纳入正常工作并对该想法的落实、跟踪和调整有明确的思路。

· 团队培训

这点相信大家没有异议，这一模式一般用在新想法的快速推广。如何快速、有效地把变革的思想和做法落实到团队，让团队中所有人都能达到统一认识水平，团队培训必不可少。通常由专职负责人起草新想法的推广方案，通过评审之后即可开展团队培训工作，方式也不一定限于传统的一对多培训方式，可以有发散和变通，如头脑风暴、焦点小组等都能起到不错的效果。

· 不妨一试

如果团队成员有人提出了一个新想法，我们通常会做出判断，想法不可行则

罢,如果可行,也不倾向马上就进行推行,因为我们知道这是一个好想法,但我们对这个新想法没有任何经验,那为了在组织中准备宣传该想法,要先把它用在自己的日常工作之中,发现和体会它的好处及其局限性。这就是“不妨一试”模式,该模式执行时间不一定会长,执行人有时候也不一定会是自己,通常选择团队中相对比较资深的成员进行短期尝试,然后根据尝试结果再做下一步计划安排,尤其对那些涉及面比较广的变化而言更是如此。

· 按部就班

“按部就班”强调做事的计划。参考敏捷开发中的迭代思想,“按部就班”就是要我们制定一个引入变化的计划,然后一步一步去实施。使用一种叠加式的策略,在保持长期目标的同时分批实现一些短期目标,这就是这个模式的思想,当然具体实施的时候需要视情况而定,对于那些比较简单的新想法而言,一步到位也很常见;但如果实施时间预估较长,那就建议“按部就班”的去推行这些变化。

另外,我们也可以使用一些辅助型工具来优化塑造研发文化的过程。如使用以“电子平台”为代表的各种工具,从而加速新想法的落地。有时候新想法已经引入但效果不一定理想,需要我们不断关注工作的进展并对实施过程中碰到的问题进行回顾总结时,可以引入“回顾时间”模式。

3. 成功的研发文化特征

通过对公司文化的利用和屏蔽,通过引入变化建设自身团队的研发文化,最终,都需要经过对比明确团队研发文化的现状和改进方向。在前人总结的基础上^[46],我们认为一个成功的研发文化应该具有以下主要特征:

(1) 互相尊重

互相尊重可能是成功的研发文化中最为重要的一个特征。一方面,团队需要对个人有尊重意识,例如,技术人员的专业性决定了其给出的设计方案和评估具有权威性,希望这种专业性和权威性能得到别人的认可。另一方面,个人也需要尊重团队,在某些团队决策确定之后,需要具备大局观,尊重团队的决定。互相尊重的团队会彼此倾听,考虑他的建议、立场和想法。这里的互相尊重不仅仅指的是研发团队内部,同时还表现为外部团队对于研发文化的一种认可。

(2) 重视创新

相较传统型行业,软件开发被认为是一种创新性工作。程序的设计和编码都是技术人员所从事的一项创造性工作,因为几乎没有什么代码是可以直接进行照搬照抄。产品演化速度非常快,决定“做什么”的产品设计的背后是技术人员“如何去做”的问题,选择最好的方式去实现产品需求,需要付出大量的思考和创新。

显然,良好的研发文化应该鼓励创新,但完全鼓励创新又不一定是一个好主意。国内软件开发的过程很多是软件应用、借鉴的过程,很多应用的方式存在标

准和最佳实践。避免“重复造轮子”也是当前软件开发过程中比较典型的一个问题。对于崇尚创新的研发文化，在需求和用户体验驱动的环境下，需要有意识的考虑创新与环境的平衡。

（3）形成标准

在团队中，一定要形成标准，标准是研发文化的一个重要组成部分。技术团队常见的标准包括设计规范、开发流程、代码质量标准、交付流程、测试用例、版本控制等多个方面，也包括文档的风格、开会的标准议程等非技术类标准，甚至包括团队成员请假邮件的标准模板等内容。

（4）崇尚沟通

只有通过充分且有效的沟通，才能确保团队每个人具有一致的思想和意识形态，也才能朝同一个目标前进。而软件开发过程中，沟通的重要性更为突出。好的研发文化一方面应该鼓励各个层级的沟通，而不是纯粹的平级之间的沟通；另一方面，也鼓励正式或非正式的沟通。对于开发人员而言，沟通通常不是强项，所以研发文化需要克服人和流程上的困难，例如敏捷开发思想下就有很多方法和工程实践来推动团队之间展开沟通。

对于跨团队之前的协作，因为沟通不畅会导致更为严重的后果。在 9.3 节的对外管理中，我们已经深入讨论了政治协商和沟通的策略，研发文化的建立也需要考虑这些策略。

（5）学习环境

研发团队应该是一个学习型团队。软件行业普遍存在的一种说法是，当前在使用的开发工具中一半以上将在两年之内被淘汰和替换，也就意味着软件开发永远是一个不断学习、不断更新的过程。另一方面，开发工作本身也是没有止境的，正如事情永远都做不完一样，代码也永远都写不完。有时候，技术人员需要放下手上正在写的代码，然后找到更好的方法、工具和框架在实现这些代码。

组织团队培训和分享，参加学术会议，或者直接推荐一本书都是促进学习比较正式的手段。当然，开展代码和设计审查也是帮助开发人员学习的好时机。

（6）具有激情

我们已经在 9.1.2 讨论了激励，而激励只是激情的一个来源。激情的另一个重要来源是开发人员自身的核心价值体现。有些开发人员对技术充满激情，而有些开发人员则更想深入业务和产品。让开发人员了解他们自身的价值所在，就能激发他们的激情。

成功的研发文化特征还包括公平、授权、职业精神等，读者可参考相关资料^[46]进行进一步学习。

10.2 作为技术管理者开展工作

很多软件公司的技术管理者往往都是技术出身，原本在技术岗位上干得不错，拥有不错的技术和业务素质，所以“技而优则仕”，走上管理岗位。但岗位的提升却也不得不面临自身工作角色和定位的问题。例如，以前只要管好自己、做好技术一件事情，现在却要管好一个团队，负责一摊子事情；以前只要听比人指挥做事就行，现在要向上级请示、与同级协作并管理下级等。工作的层次和定位决定着工作的开展方式，技术管理者想要提升工作效率，明确自身工作的层次和定位至关重要。

10.2.1 工作的层次和定位

1. 组织结构层次模型

在第 1 章中我们讨论并明确了技术管理者的角色，本章中我们再来分析管理者在一个组织结构中的位置。图 10-4 描述的是经典的金字塔形组织结构模型图，从这张图可以看到组织结构表现为一种三层结构，这三层分别称为战略层、规划层和执行层。每一层对应不同的角色，战略层对应决策者角色，规划层对应管理者角色，而执行层对应执行者角色。决策者是整个组织的领导者，是最先领路的人；管理者是推动事情发展的人；而执行者则是实际执行的人。

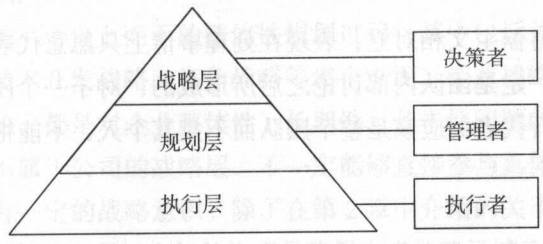


图10-4 三层组织结构模型

决策者、管理者和执行者构成了一个组织的基本结构，本书讨论的对象就是位于规划层的管理者角色。管理者是组织的中坚力量，在带领下属团队完成本部门工作任务的同时，也要接受上级的领导，同时兼有下属和领导的双重身份。

另一方面，不同的层次决定不同的工作重点，如果我们把工作类型分成日常型工作、分析型工作和创新型工作，那么每个层级的工作重点如表 10-3 所示。我们可以看到对于管理者而言，所应该从事的是分析型工作，也即更多的时间和精力应该放在分析问题、提供问题的解决方案并协调相关资源的活动上。

表10-3 不同层次的工作重点

类别	执行层	规划层	战略层
日常型工作	70%：遵守规定	20%：监控日常工作	10%：验收日常工作
分析型工作	20%：发现并报告问题	60%：分析查找问题根源，提出解决方案和所需资源	20%：评估方案，提供资源
创新型工作	10%：日常工作中利用新方法	20%：复制创新工作并转为工作流程和规范	70%：新的思路、方法、战略规划

2. 技术管理者的定位问题

梳理技术管理者角色的定位并不困难，明确定位并执行才是关键。在现实的研发过程中，由于定位所导致的管理问题也屡见不鲜。这些问题有些会导致技术管理者个人难以开展工作，但更多的时候是影响团队气氛和工作效率。一些常见的由于定位所引发的管理问题包括但不限于：

· 官僚主义

官僚主义在任何组织或多或少都存在，有些管理者会过分看重自己的级别，优越感强烈并且自我感觉良好。对于产品研发中的业务范围、进度等都是自己说了算，不重视、甚至不相信技术团队的集体智慧和能力。

· 个人主义

个人主义和官僚主义相对立，表现在处理事情上只愿意代表个人而不是一个团队。团队意见一定是团队内部讨论之后所形成的，对于一个团队管理者而言，所想、所言、所行代表的应该是整个团队而不是其个人，不能把自己当作是一名普通的执行者。

· 传话筒

管理者处于金字塔三层模型中的中间一层，需要起到承上启下的作用，而不仅仅只是把上层的决策传递给下层，再把下层的执行结果反馈给上层。避免形成传话筒就需要对信息进行过滤和处理，而不是简单的流转。

· 向下和向上管理

过于向上管理和过于向下管理都存在管理风险。从技术岗位转型为管理岗位的技术人员经常忽视向上管理的重要性。有些技术管理者深得下属信赖和拥戴，于是处处站在下属的立场向上级提出某些不合理的要求。而不是技术出身的管理人员则倾向于完全站在产品和业务的角度去看待技术人员，把技术本身以及技术团队当作仅仅是实现业务的手段和工具。

对于这些定位问题，技术管理者一方面要正视自我，掌握本书其他章节中介

绍的方法和技巧，正确面对挫折和痛苦。另一方面，也要明确自己作为一个推动者的定位，作为推动者开展工作。

10.2.2 作为推动者开展工作

作为推动者，最主要的作用在于提升效率。推动者是行动的发起者，自发性强，目的明确，有高度的工作热情和成就感。明白技术管理者的日常工作内容有助于理解技术管理者转型过程中可能会碰到的问题，在进行转型之前，我们可以分析和挖掘作为一名技术管理者的工作开展方式，以及抱着这些问题再次回到本书的开头回顾转型之路。图 10-5 梳理了作为一名技术管理者所从事的六大类典型工作以及每一类工作中的推进作用。

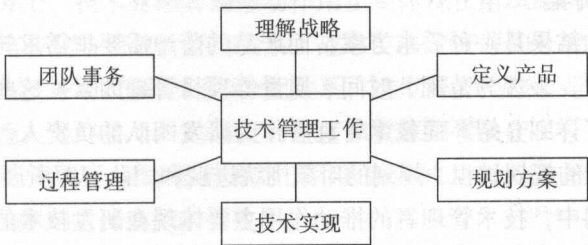


图10-5 技术管理者的工作

(1) 理解战略

企业的战略是一个自上而下的整体性规划过程，其中包括营销战略、品牌战略、融资战略、技术开发战略、竞争战略等多个方面。企业战略虽然有多种，但基本属性是相同的，都是对企业整体性、长期性、基本性问题的计谋。技术管理者在组织架构上不属于公司的战略层，不一定能够直接参与具体战略的规划和设计，但同样需要有一定的战略意识。除了在第 2 章中介绍的关于行业分析的通用方法之外，与上层管理者保持良好的沟通是确保自身战略敏感度的有效方法。

在战略层面，技术管理者的推动作用主要体现在战略目标的分解和宣传。当公司层面的战略目标确定之后，如何让团队中的所有人都能理解和吸收这个目标，并转化为日常工作中的动力是一件并不容易的事情。战略自上而下的特性决定了低层执行者并不一定能够充分理解和把握上层的思想，从而无法明确工作的目的以及提升自身的激情。技术管理者推动者有助于缓解这一普遍存在的现象。

(2) 定义产品

产品定义是指确定产品需要做哪些事情，通常采用产品需求文档来进行描述。需求文档中可能包括的信息有产品的愿景、目标市场、竞争分析、产品功能的详细描述、产品功能的优先级、产品用例、系统需求、性能需求和销售及支持需求等。

通常，不同的组织中产品定义的工作也会由不同的人来组织，产品负责人或运营负责人是该类工作的主要执行者，但技术管理者也会在很多场合深入参与这个过程。正如第3章中介绍的业务结构和产品平台策略，技术管理者需要从技术预测、产品线和产品平台规划角度出发考虑产品的定义。

在产品定义上，技术管理者的推动作用主要体现在产品需求文档的确定和评审。产品需求的确定包括大方向上的愿景和策略，也包括具体产品功能的规划和设计。定义的过程也可以采用迭代方式演进，每一个阶段性的产品定义成果在正式交付前都需要技术的介入。目前有些类如视频直播、机器学习等新型产品形态，技术在产品定义过程中可能还会起到反哺作用，这就需要技术管理者从技术角度给出专业的意见以便更好地推动产品设计。

（3）规划方案

产品定义的结果是一种需求方案，而产品的落地需要把需求转化为技术人员能够实现的过程，表现为范围、时间、质量等项目管理因素。这些因素我们已经在第7章中做了详细介绍。而技术管理者作为研发团队的负责人，通常都需要参与项目实施方案的规划过程，规划的对象围绕进度和团队资源所展开。

在方案规划中，技术管理者的推动作用主要体现在研发技术的制定和落地。技术解决方案作为整体研发方案的一部分，需要作为单独的一个步骤体现在研发计划中，并遵循一定的评审机制以形成过程资产。而研发的进度和资源安排，通过可视化的视图和工具展现在研发团队面前也属于技术管理者的工作内容。

（4）技术实现

研发计划制定之后就进入开发阶段。开发阶段一般涉及开发人员的集中式投入，以及系统实现技术体系的建立。我们在第4章和第5章中分别介绍了目前主流的技术开发思想和模式、架构设计的考虑点。视不同的公司环境，技术管理者参与技术实现工作的程度也会有所不同，而第6章中提到的技术创新工作也根据行业和业务体系会表现出不同的创新形式。

在技术实现中，技术管理者的推动作用主要体现在系统架构的设计和演进。系统架构一般都会经历从简单到复杂、从复杂到难以维护、再从难以维护到拆分成简单的循环过程，这一过程中随着架构的演进，需要进行合适的技术选型，以及推进重构等工程实践的落地。

（5）过程管理

技术管理者需要对研发结果负责，而正确的结果来自于正确的过程，研发方案的实施体现为一系列的过程。研发过程体系的建立属于第8章中的内容，但实际上任何软件开发相关的工作都可以归为一种过程。对于组织级别的过程建设，技术管理者可以给出自身的建议和意见；而对于研发内部，技术管理者则是过程

管理的主要责任人。

在过程管理中，技术管理者的推动作用主要体现在过程体系的建立和改进。业界存在一些主流的过程管理方法和框架，也有一些成熟的工程实践可以直接应用。技术管理者可以参考并裁剪这些过程以形成符合当前团队的过程体系。同时，过程管理的本质还是在于过程改进，需要在快速变化的行业、产品和环境中调整过程。

(6) 团队事务

团队的工作任务分解和组织结构确立、人员的招聘、培训和绩效考核都属于技术管理者的日常事务。我们在第 9 章对团队组织管理的各个方面做了展开，包括向下管理、向上管理、向外管理和自我管理。团队事务分散在以上各个工作环节中。

在团队事务上，技术管理者的推动作用主要体现在组织结构的形成和优化。组织结构与公司的战略规划有直接关系，战略层决定资源的分配，管理层确定组织的效率并决定任务与资源之间的分配关系，这里需要根据公司战略考虑成本、收入之间的平衡。最终各个产品研发的进度和资源因素决定了任务开展的效率，如图 10-6 所示。所以，技术管理者对于团队事务的处理实际上影响着公司的整个战略，同时也影响着团队的工作效率。

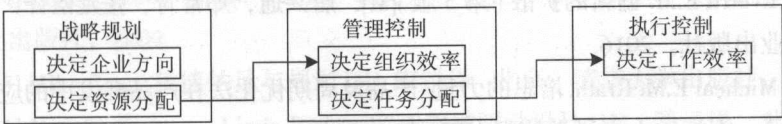


图 10-6 组织结构的确定

以上是作为一个技术管理者所从事工作中最重要的六大方面内容，其他还有可能会涉及的部分财务管理、营销管理等现代企业管理方面的内容请参考相关书籍^[47]，这里不做展开介绍。

10.3 本章小结

本章作为全书总结，围绕如何成为一名合格的技术管理者这一话题进一步提出转型的最重要切入点是意识形态的转变。意识形态的转变首先是思维模式的建立，技术管理者有了自身的思维方式之后就能对目前存在的问题提出自身的想法，从而尝试通过引入变化的方式改变现状，最终建立高效的研发文化。

在任何组织中都存在固定的结构层次模型，明确在组织中的定位之后，技术管理者需要作为一个推动者开展工作。本章最后还对技术管理者日常工作做了梳理。

参考文献

- [1] 郑天民. 系统架构设计：程序员向架构师转型之路 [M]. 北京：人民邮电出版社，2017.
- [2] Henry Mintzberg. 管理工作的本质（经典版）[M]. 方海萍译. 浙江：浙江人民出版社，2017.
- [3] ISO/IEC 9241-11 Ergonomic Requirements for Office Work with Visual Display Terminals (VDT) - Part II Guidance on Usability. 1998; ISO/IEC 9241-11; 1998 (E)
- [4] www.wq usability.com/articles/getting-started.html
- [5] http://semanticstudios.com/user_experience_design/
- [6] Everrtt E.M. 创新的扩散（第5版）[M]. 唐兴通，郑常青，张延臣译. 北京：电子工业出版社，2016.
- [7] Micheal E.McRrath. 培思的力量：产品及周期优化法在产品开发中的应用[M]. 徐智群译. 上海：上海科学技术出版社，2004.
- [8] 梁飞. 框架设计原则 .ppt, 2012.
- [9] Frank Buschmann, Kevin Henney, Douglas C. Schmidt. 面向模式的软件架构：分布式计算的模式语言（卷4）. 肖鹏，陈立译. 北京：人民邮电出版社，2010.
- [10] Erich Gamma, Richard Helm, Ralph Johnson. 设计模式：可复用面向对象软件的基础 [M]. 刘建中等译. 北京：机械工业出版社，2007.
- [11] Robert C.Martin. 敏捷软件开发：原则模式与实践. 邓辉译. 北京：清华大学出版社，2003.
- [12] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. 面向模式的软件架构·卷1：模式系统 [M]. 袁国忠译. 北京：人民邮电出版社，2013.
- [13] Kirk Knoernschild .Java 应用架构设计：模块化模式与 OSGi[M]. 张卫滨译. 北京：机械工业出版社，2013.
- [14] <https://www.javacodegeeks.com/2015/04/microservice-design-patterns.html>

- [15] Philippe Kruchten. Rational 统一过程引论（第二版·影印版）[M]. 北京：中国电力出版社，2003.
- [16] Nick Rozanski, Eoin W. 软件系统架构：使用视点和视角与利益相关者合作（第二版）[M]. 侯伯薇译. 北京：机械工业出版社，2013.
- [17] Eric Evans. 领域驱动设计：软件核心复杂性应对之道 [M]. 赵俐，盛海艳，刘霞译. 北京：机械工业出版社，2010.
- [18] Gregor Hohpe, Bobby Woolf. 企业集成模式 [M]. 荆涛，王宇译. 北京：中国电力出版社，2006.
- [19] http://www.melconway.com/Home/Conways_Law.html?spm=5176.100239.blogcont8611.8.6je70H
- [20] https://en.wikipedia.org/wiki/CAP_theorem
- [21] <http://dl.acm.org/citation.cfm?doid=1394127.1394128>
- [22] James M. Utterback, William J. Abernathy. A Dynamic Model of Product and Process Innovation, Omega[J] 1975, 3(6): 639-656
- [23] Grady Booch, James Rumbaugh, Ivar Jacobson. UML 用户指南（第2版·修订版）[M]. 邵维忠，麻志毅，马浩海等译. 北京：人民邮电出版社，2013.
- [24] James Shore, Shane Warden. 敏捷开发的艺术 [M]. 王江平等译. 北京：机械工业出版社，2009.
- [25] Mike Cohn. 敏捷估计与规划 [M]. 宋锐译. 北京：清华大学出版社，2007.
- [26] Vaughn Vernon, Linda Rising. 实现领域驱动设计 [M]. 滕云译. 北京：电子工业出版社，2014.
- [27] https://en.wikipedia.org/wiki/Delphi_method
- [28] https://en.wikipedia.org/wiki/COSMIC_software_sizing
- [29] Tom DeMarco, Timothy Lister. 与熊共舞 [M]. 钱一一译. 台湾：经济新潮社，2004.
- [30] Frederick P. Brooks, Jr. 人月神话 [M]. 汪颖 译. 北京：清华大学出版社，2015.
- [31] Kenneth Rubin. Scrum 精髓 敏捷转型指南 [M]. 姜信宝，米全喜，左洪斌译. 北京：清华大学出版社，2014.
- [32] Mary Poppendieck. Lean Software Development: An Agile Toolkit [M]. Addison-Wesley Professional, 2003.
- [33] Jeffrey Liker. 丰田模式：精益制造的 14 项管理原则 [M]. 李芳龄译. 北京：机械工业出版社，2016.
- [34] David J. Anderson. 看板方法：科技企业渐进变革成功之道 [M]. 章显洲译.

武汉：华中科技大学出版社，2014-02-01.

[35] <http://www.sei.cmu.edu/cmmi/>

[36] Esther Derby, Esther Derby. 敏捷回顾：团队从优秀到卓越之道 [M]. 周全，冯左鸣，拓志祥，李丽森译. 北京：电子工业出版社，2012.

[37] Project Management Institute. 项目管理知识体系指南（第5版）[M]. 许江林等译. 北京：电子工业出版社，2013.

[38] Gerald M. Weinberg. 成为技术领导者：掌握全面解决问题的方法 [M]. 余晟译. 北京：电子工业出版社，2015.

[39] Maslow A.H. 马斯洛论管理 [M]. 邵冲，苏曼译. 北京：机械工业出版社，2013.

[40] Douglas M. Mc Gregor. 企业的人性面 [M]. 李宙，章雅倩译. 长春：北方妇女儿童出版社，2017.

[41] Frederick Herzberg. 赫茨伯格的双因素理论 [M]. 张湛译. 北京：中国人民大学出版社，2016.

[42] https://en.wikipedia.org/wiki/DISC_assessment

[43] Peter F. Drucker. 管理的实践 [M]. 齐若兰译. 北京：机械工业出版社，2009.

[44] Mary Lynn Manns, Linda Rising. 拥抱变革：从优秀走向卓越的48个组织转型模式 [M]. Evelyn Tian 译. 北京：清华大学出版社，2014.

[45] https://en.wikipedia.org/wiki/Tuckman%27s_stages_of_group_development

[46] Mickey W. Mantle, Ron Lichty. 告别失控：软件开发团队管理必读 [M]. 赵普明，黄倩，张维维，钱昊译. 北京：人民邮电出版社，2016.

[47] 郭适融. 现代企业管理：理念、方法、技术（第二版）[M]. 北京：清华大学出版社.

非卖品！！严禁（售卖和上传互联网平台）！！违者责任自负！！


责任编辑：毛俊宁

装帧设计： 北京中尚图
文化传播有限公司

向技术管理者

转型

From Programmers to
Technical Managers



本书主要包含软件开发人员如何向技术管理者进行转型的一些思路、方法和工程实践，包括转型过程中所涉及的关于行业、技术和管理三大知识体系以及意识形态的转变和提升等内容。深入剖析成为一名合格的技术管理者所需要的各项软硬技能，重点对目前业界主流的互联网行业下所需掌握的产品开发、技术架构和技术创新领域，以及作为一名技术管理人员所需具备的组织和过程管理能力进行详细展开，并结合一些典型的场景和案例进行分析，帮读者了解并掌握迈向技术管理者所需的各种知识体系和实践技巧。

本书面向立志于转型成为技术管理岗位的软件开发人员，读者不需要有很深的技术水平，也不需要具体的开发团队管理经验，但熟悉软件开发整体流程有助于更好地理解书中的内容。同时，也可以供不同产业、不同开发模式下的技术管理者同行参考，希望能给日常研发和管理工作带来启发和帮助。

上架建议：企业管理

ISBN 978-7-5108-6316-5



9 787510 863165 >

定价：59.00 元